Scalable and Privacy-preserving Off-chain Computations

vorgelegt von
M.Sc.
Jacob Eberhardt

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. rer. nat. Florian Tschorsch
Gutachter:	Prof. Dr.-Ing. Stefan Tai
Gutachter:	Prof. Andrew Miller, Ph.D.
Gutachter:	Prof. Dr.-Ing. Stefan Schulte

Tag der wissenschaftlichen Aussprache: 23. April 2021

Berlin 2021

Abstract

Blockchains are distributed systems that allow mutually distrusting parties to process transactions in a censorship-resistant way while establishing an immutable transaction history without trusting a third party. These properties, however, do not come for free. Unlike other large scale distributed systems, blockchains do not scale. They suffer from low transaction throughput and high costs resulting from redundant transaction processing and consensus overhead. Furthermore, there is no privacy protection in blockchain networks: All transaction data is necessarily exposed to the network for independent validation, essentially making it public.

In this thesis, we introduce off-chaining to address the privacy and scalability challenges faced by today's blockchains: Instead of technically modifying blockchains themselves, we propose to move computations and data off the blockchain — without compromising its desirable properties in the process. Off-chaining reduces the work a blockchain has to perform and improves its privacy properties by avoiding publishing data in the first place. As a first contribution, we identify off-chaining patterns that can be instantiated in the context of blockchain-based applications and provide solutions to recurring design problems. As a second contribution, we provide an in-depth analysis and comparison of off-chain computation approaches, which represent a particularly powerful privacy- and scalability-engineering abstraction. We identify zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs), a class of cryptographic protocols, as the most suitable approach.

Developers, however, are ill-equipped to instantiate zk-SNARK-based off-chain computations to address blockchain-based applications' privacy and scalability needs. Their instantiation is complex and error-prone; suitable programming abstractions and software tools are missing. To bridge this gap, we present ZoKrates, the first language and toolbox for zk-SNARK-based verifiable off-chain computations that allows non-expert developers to specify and execute off-chain computations in a usable and efficient manner as our third contribution. As our fourth contribution, we demonstrate the viability of ZoKrates, and more generally, off-chaining, to address privacy and scalability concerns in an extensive evaluation in the context of three relevant blockchain-based applications: decentralized energy trading, scalable blockchain relays, and privacy-preserving token transfers. Beyond these use cases, the open-source software that originated in the context of this thesis has found independent application in academia and industry.

Zusammenfassung

Blockchaintechnologien erlauben sich gegenseitig misstrauenden Akteuren zensurresistent Transaktionen in einem verteilten System zu verarbeiten und dabei eine unveränderliche Transaktionshistorie zu etablieren, ohne hierfür eine vertrauenswürdige dritte Partei hinzuzuziehen. Allerdings stehen diese Eigenschaften mit anderen wünschenswerten Qualitätseigenschaften verteilter Systeme in Konflikt. Blockchains skalieren nicht und Konsensfindung sowie redundante Transaktionsverarbeitung führen zu hohen Verarbeitungskosten und niedrigem Durchsatz. Zudem sind private Informationen nicht geschützt; alle transaktionsrelevanten Daten werden im Rahmen ihrer Verarbeitung netzwerköffentlich.

Um diese Probleme bezüglich Skalierbarkeit und Datenschutz zu adressieren, führt diese Arbeit Off-chaining als Kernkonzept ein: Off-chaining bezeichnet das Auslagern von Daten und Berechnungen auf Blockchain-externe Ressourcen, ohne dabei deren Schlüsseleigenschaften zu kompromittieren. Ohne das Blockchainsystem selbst zu modifizieren, reduziert Off-chaining dessen Validierungslast und erlaubt die externe Speicherung und Verarbeitung sensibler Informationen. Im Rahmen dieser Arbeit werden Off-chaining Patterns identifiziert, die im Kontext Blockchain-basierter Applikationen instanziiert werden können und wiederkehrende Designprobleme lösen. Hierbei stellen insbesondere Off-chain Berechnungen eine mächtige, jedoch wenig untersuchte Ausprägung dar. Die Analyse verschiedener Ansätze zur Realisierung solcher Berechnungen identifiziert zero-knowledge Succint Non-interactive Arguments of Knowledge (zk-SNARKs), eine Klasse kryptographischer Protokolle, als vielversprechendste Ausprägung.

Allerdings ist die Instanziierung zk-SNARK-basierten Off-chainings äußerst komplex und somit wenigen Experten vorbehalten. Geeignete Programmierabstraktionen und softwaretechnische Werkzeuge fehlen. Um dieses Problem zu adressieren, präsentieren wir ZoKrates, die erste Programmiersprache und Sammlung von Softwarewerkzeugen für zk-SNARK-basierte verifizierbare Off-chain Berechnungen. ZoKrates erlaubt es Entwicklern Off-chain Berechnungen auf nutzerfreundliche Art zu spezifizieren und effizient auszuführen, ohne spezifische Expertise in Kryptographie und deren Anwendung zu benötigen. Im Rahmen einer ausführlichen Evaluation wird die praktische Signifikanz von ZoKrates, und Off-chaining, durch die exemplarische Anwendung auf drei relevante blockchain-basierte Applikationen demonstriert, welche sich mit Skalierbarkeits- oder Datenschutzproblemen konfrontiert sehen: dezentraler Energiehandel, Blockchain-Relays und anonyme Token-Transfers. Die im Kontext dieser Arbeit entstandenen Softwarelösungen fanden darüber hinaus unabhängige Anwendung in Wissenschaft und Industrie.

Acknowledgements

This thesis could not have been completed without the support of many brilliant minds who inspired, influenced, and supported me over the last years.

First of all, I would like to thank my supervisor Prof. Dr. Stefan Tai. He supported me throughout my work on this thesis, offered valuable advice, always believed in my abilities, and gave me the opportunity and freedom to work in an exciting and novel field of computer science. I will always uphold his research ideals of leveraging deep technical expertise to address problems with relevance in the real world.

I thank the reviewers Prof. Andrew Miller, Ph.D., and Prof. Dr. Stefan Schulte for their valuable feedback on this thesis as well as a fair defense. Furthermore, I thank Prof. Dr. Florian Tschorsch who served on the doctoral committee as chairman.

Without Prof. Dr. David Bermbach, who supervised my bachelor's and master's theses and introduced me to computer science research, I would probably not have pursued a Ph.D. I'm deeply thankful for his encouragement and always enjoyed working together.

I would like to thank my colleagues, Dr. Anselm Busse, Maria Borges, Dominik Ernst, Jonathan Heiß, Dr. Markus Klems, Jörn Kuhlenkamp, Dr. Frank Pallas, Marco Peise, Max-Robert Ulbricht, and Sebastian Werner. I will not only remember you as great coworkers but as friends. Further thanks go to Anita Hummel for her excellent administrative support.

Over the course of my work, I had the privilege to collaborate with many exceptional students. I am particularly grateful to have worked with Dennis Kuhnert, Steffen Härtlein, and Paul Etscheit. I always enjoyed our discussions and your support was much appreciated.

Additionally, I would like to thank Dr. Christian Reitwießner for his support of the research efforts that later became ZoKrates, Thibaut Schaeffer for his friendship and significant contributions to the ZoKrates open source project, and the Ethereum Foundation for their ongoing support.

Finally, I want to thank my parents, my brother, my family and of course Amelie. Without your love and support, this work would not have been possible.

Jacob Eberhardt
Berlin, April 2021

Contents

I Foundations **1**

1 Introduction **3**
1.1 Problems . . . 4
1.1.1 Peer-to-peer Energy Trading . . . 6
1.1.2 Efficient Blockchain Relays . . . 6
1.1.3 Token Transfers . . . 7
1.2 Research Questions . . . 7
1.3 Contributions . . . 8
1.3.1 Off-chaining Patterns for Privacy and Scalability . . . 8
1.3.2 Identification and Comparison of Off-chain Computation Approaches . 9
1.3.3 Framework for Scalable and Privacy-preserving Off-chain Computations 10
1.3.4 Privacy-preserving and Scalable Variants of Decentralized Applications 10
1.4 Published Material . . . 11
1.5 Thesis Organization . . . 12

2 Background **15**
2.1 Blockchains and Smart Contracts . . . 15
2.1.1 Bitcoin . . . 16
2.1.2 Ethereum and Smart Contracts . . . 17
2.1.3 Decentralized Applications . . . 18
2.1.4 Private and Permissioned Blockchains . . . 19
2.2 Verifiable Computation . . . 20
2.2.1 Foundations of Verifiable Computation Schemes . . . 21
2.2.2 Overview of Verifiable Computation Schemes . . . 24

3 Related Work **29**
3.1 Off-chaining . . . 29
3.1.1 Off-chaining Patterns . . . 30
3.1.2 Off-chain Computations . . . 30
3.2 zk-SNARK-based Off-chain Computations . . . 30

3.2.1 Applications of zk-SNARK-based Off-chain Computations 31
3.2.2 zk-SNARK-based Frameworks for Private Smart Contracts 31
3.2.3 Specification of Verifiable Off-chain Computations 32
3.3 Conclusion 34

II Off-chaining **35**

4 Off-chaining Patterns **39**
4.1 Off-chaining Definition 39
4.2 Motivational Examples 40
4.2.1 Fair Chess 41
4.2.2 Blockchain-based Service Marketplace 41
4.3 Off-chaining Patterns 42
4.3.1 Content-addressable Off-chain Storage Pattern 43
4.3.2 Verifiable Off-chain Computation Pattern 45
4.3.3 Off-chain Signatures Pattern 48
4.3.4 Optimistic Finalization Pattern 50
4.3.5 Low Contract Footprint Pattern 53
4.4 Impact and Open Questions 55
4.4.1 Scalability 55
4.4.2 Privacy 56
4.5 Conclusion 57

5 Approaches to Off-chain Computations **59**
5.1 Off-Chain Computations 60
5.1.1 Definition 60
5.1.2 Basic Architecture 60
5.1.3 Design Space 61
5.2 Off-Chain Computation Approaches 63
5.2.1 Verifiable Off-chain Computations 63
5.2.2 Enclave-based Off-chain Computations 66
5.2.3 Incentive-driven Off-chain Computations 67
5.2.4 Secure Multi-party Computation-based Off-chain Computations 70
5.2.5 Realizations 71
5.3 Comparative Assessment 72
5.3.1 Scalability 74
5.3.2 Privacy 74
5.3.3 Security 75
5.3.4 Programmability 75
5.4 Conclusion 76

III ZoKrates **77**

6 Design Goals and Overview **81**
6.1 Motivation & Design Goals . . . 81
6.1.1 Usability . . . 82
6.1.2 Generality . . . 82
6.1.3 Efficiency . . . 82
6.2 ZoKrates Overview . . . 83
6.2.1 Program Specification . . . 83
6.2.2 Compilation . . . 84
6.2.3 Setup . . . 85
6.2.4 Verification Contract Generation & Deployment . . . 85
6.2.5 Off-chain Program Execution . . . 87
6.2.6 Proving Correct Execution . . . 87
6.2.7 On-chain Proof Verification . . . 89

7 ZoKrates Intermediate Representation **91**
7.1 Prime Fields and Arithmetics . . . 92
7.2 Provable Abstractions . . . 94
7.2.1 Arithmetic Circuits . . . 95
7.2.2 Straight-line Programs . . . 97
7.2.3 Rank-1 Constraint Systems . . . 99
7.3 Expressiveness and Efficiency . . . 103
7.3.1 Provable Computations . . . 103
7.3.2 Proving Computations Efficiently . . . 105
7.4 The ZoKrates Intermediate Representation . . . 110
7.5 Conclusion . . . 113

8 The ZoKrates Language **115**
8.1 Challenges and Design Principles . . . 115
8.1.1 Domain-specific Challenges . . . 117
8.1.2 Design Principles . . . 117
8.2 Syntax and Semantics . . . 119
8.2.1 Program Structure and Fundamentals . . . 119
8.2.2 Types . . . 121
8.2.3 Operators . . . 127
8.2.4 Control Flow . . . 134
8.2.5 Modules and Imports . . . 137
8.3 The ZoKrates Standard Library . . . 139
8.3.1 Utility Functions . . . 139
8.3.2 Elliptic Curve Cryptography . . . 140
8.3.3 Digital Signatures . . . 141
8.3.4 Hashing . . . 141
8.4 Compilation . . . 143

8.4.1 Lexical Analysis . . . 144
8.4.2 Syntactic Analysis . . . 145
8.4.3 Semantic Analysis . . . 145
8.4.4 Unrolling . . . 146
8.4.5 Flattening . . . 148
8.4.6 Propagation . . . 149
8.4.7 Intermediate Code Generation . . . 149
8.4.8 ZIR Optimization . . . 150
8.4.9 Constraint Generation . . . 151
8.4.10 Backend-specific Optimization . . . 151
8.5 Conclusion . . . 151

9 ZoKrates Architecture **153**
9.1 Command-Line Interface . . . 155
9.2 Compiler . . . 155
9.3 Interpreter . . . 156
9.4 Application Binary Interface En-/Decoder . . . 156
9.5 Backends . . . 160
9.6 Constraint Converter . . . 161
9.7 Contract Exporter . . . 162

10 Implementation **165**
10.1 Organization of the Codebase . . . 165
10.2 Parser . . . 167
10.3 Backends . . . 168
10.4 Embeds . . . 169
10.5 Directives & Solvers . . . 169
10.6 WebAssembly and ZoKrates JavaScript Binding . . . 170
10.7 ZoKrates Remix Plugin . . . 170
10.8 Continuous Integration and Delivery . . . 171

11 Performance Evaluation **173**
11.1 Off-chain Steps . . . 175
11.1.1 Repeated Steps . . . 176
11.1.2 One-time Steps . . . 178
11.1.3 Compiler Optimization Impact . . . 180
11.2 On-chain Steps . . . 183
11.2.1 Verification . . . 185
11.2.2 Deployment . . . 187
11.3 Conclusion . . . 190

IV Applications **193**

12 Privacy-Preserving Energy Trading **197**
12.1 Motivation and Overview . . . 197
12.2 Background . . . 199
12.2.1 Energy Grid Organization . . . 199
12.2.2 Energy Market Organization . . . 199
12.2.3 Prosumer Economics . . . 200
12.3 Related Work . . . 201
12.3.1 Blockchain-based Energy Trading . . . 201
12.3.2 Privacy in Smart Grids . . . 201
12.3.3 Privacy in Blockchain-based Energy Trading . . . 202
12.3.4 Self-sufficient Microgrids . . . 202
12.4 System Design . . . 202
12.4.1 A Blockchain-based Architecture . . . 202
12.4.2 Adding Privacy . . . 204
12.5 ZoKrates-based Implementation . . . 206
12.5.1 Smart Meter . . . 206
12.5.2 Household Processing Unit . . . 207
12.5.3 Netting Entity . . . 207
12.5.4 Blockchain . . . 209
12.6 Netting . . . 209
12.6.1 Characterizing Desirable Netting Results . . . 209
12.6.2 A Fair and Constructive Netting Algorithm . . . 212
12.7 Performance Evaluation . . . 215
12.8 Conclusion . . . 216

13 Scalable Blockchain Relay **217**
13.1 Motivation and Overview . . . 217
13.2 Conceptual Design . . . 219
13.2.1 Off-chain Validation of a Single Block Header . . . 219
13.2.2 Off-chain Validation of Epoch Batches . . . 221
13.2.3 Simplified Payment Verification for Intermediary Headers . . . 222
13.2.4 Flexible Batch Sizes . . . 223
13.3 ZoKrates-based Implementation . . . 225
13.3.1 ZoKrates Off-chain Programs . . . 225
13.3.2 Performance Evaluation . . . 226
13.4 Related Work . . . 227
13.5 Conclusion . . . 229

14 Anonymous Token Transfers **231**
14.1 Motivation . . . 231
14.2 Protocol Design . . . 232
14.2.1 Mint . . . 233

14.2.2 Transfer 234
14.2.3 Burn 235
14.3 ZoKrates-based Implementation 236
14.3.1 Off-chain Steps 236
14.3.2 On-chain Steps 237
14.4 Conclusion 237

V Conclusions **239**

15 Summary **241**

16 Outlook and Discussion **243**

A ZoKrates Grammar **245**

B Selected Supervised Theses **249**

List of Publications **251**

Bibliography **253**

List of Abbreviations **277**

List of Figures **279**

List of Tables **281**

List of Listings **283**

Part I

Foundations

CHAPTER 1

Introduction

Blockchain technologies represent a novel category of distributed systems that allow mutually distrusting parties to process transactions without having to agree on a trusted third party [319, 355]. Transaction processing is fully transparent, manipulation-safe, and censorship-resistant. Past transactions are recorded immutably and establish an agreed-on shared history. Not only have blockchains become an active area of research in computer science as well as economics, but their potential impact has inspired many other communities and industries that seek to leverage the new properties previously introduced.

Initially, blockchains were introduced with Bitcoin [258], a decentralized peer-to-peer cash system, in 2008. This proposal combined distributed system design principles with economic incentives to create a payment system that does not rely on trusted intermediaries and enables non-duplicable digital assets. As a major building block, Bitcoin introduced the blockchain as a replicated append-only data structure recording transactions ordered by consensus in a network of peers in an immutable way. Unlike in previous systems, every node participating in the payment network acts as an independent validator of all transactions and is economically incentivized to contribute to its security.

However, it soon turned out that Bitcoin's support for other desirable use cases is insufficient. Proposals like Colored Coins [19], Namecoin [259], and the idea of autonomously executing legal contracts [317] showed that implementing use cases unrelated to payments is difficult and requires forking into separate networks. This motivated general-purpose blockchain platforms like Ethereum [74, 350], which widen Bitcoin's narrow scope on financial transactions. They allow the decentralized execution of generic user-defined programs, so-called smart contracts, on a general-purpose blockchain that acts as a decentralized compute platform.

Enabled by these new platforms and driven by the insight that blockchains can not only contain financial transaction information but essentially record any data in an agreed-on and immutable way, a broad range of use cases capitalizing on these novel properties were discovered: Blockchain-based land registers independently record ownership rights [308], certificates of product authenticity or provenance can be recorded in a tamper-proof way [280, 312], self-sovereign

digital identities can be established [108, 229], and automatically executing tamper-proof insurance policies [256] can be implemented.

Applications implementing such use cases rely on decentralized infrastructure as a key part of their architecture but also comprise off-chain components, such as user interfaces and storage [313]. To indicate this composition, they are often referred to as decentralized applications (dApps) [74, 284].

In summary, both relevant use cases and generic blockchain platforms supporting decentralized applications have been developed. Yet, until today, there are few blockchain-based applications deployed and used in practice at a significant scale.

1.1 Problems

Users of applications rarely care about decentralization itself [15]. It is an abstract value that does not directly increase their economic utility. Reliability, minimal trust requirements, low costs, and prevention of lock-in effects, in turn, are direct benefits that could come through the adoption of blockchains as an application platform. However, beyond legal and economic challenges, two core technical challenges prevent these benefits from materializing and hinder blockchains' mainstream adoption.

The first challenge is scalability. In current blockchain networks, the throughput does not increase with the number of nodes in the network. Blockchains do not scale. Throughput is low, transaction costs and processing latencies are high: Bitcoin currently processes 7 transactions/sec and blocks, which confirm a batch of transactions, are created every 10 minutes on average [48]; Ethereum processes up to 25 transactions/sec and has a 15 seconds average block interval [77]. The payment processor Visa, in comparison, processes ~1700 transactions/sec on average, which are confirmed within seconds [331]. This is a fundamental disadvantage for dApps competing with traditional services, which do not suffer from such restrictions.

The scalability issue is well recognized in research [92], and there are many proposals to increase throughput and reduce latency: Novel consensus mechanisms [23], sharding [337], sidechains [359], optimistic execution approaches [135, 320], and payment channels [168] address scalability at different layers of abstraction and partially for specific use cases. Nevertheless, none of these proposals lead to the desirable level of performance yet. Scaling blockchain infrastructure itself is recognized as an intrinsically hard problem [92].

The second challenge is privacy.[1] Beyond performance, privacy is a key property computing infrastructure needs to account for: Users, for example, expect application providers to guarantee

[1] Different notions of the term *privacy* exist in the blockchain community, the privacy-engineering community, and the community of legal experts. A narrow legal definition refers to privacy as the ability to protect personal data, i.e., any information relating to an identified or identifiable natural person [285]. To better capture our intent, we follow a more general definition for the remainder of this thesis: We define privacy as the ability to protect one's sensitive information, i.e., to ensure confidentiality in addition to the protection of personal information. In the context of blockchain transactions, this definition reduces to the ability to protect transaction information as well as the transacting parties' identities.

adequate protection of their health status, shopping history, and financial information. Similarly, business owners may need to protect their supply chain information to maintain a competitive advantage. More generally, an individual's willingness to share information as well as said information's value directly depends on who has and who does not have access to it - a property thus crucial to control. At the latest since the General Data Protection Regulation (GDPR) [285] is in place, privacy protection has evolved to a mandatory requirement that needs to be fulfilled by digital infrastructure.

However, this requirement is fundamentally in conflict with the current mode of operation in modern blockchain networks [92]: In current blockchain networks, all nodes redundantly execute every single transaction. This mode of operation is fundamentally required to ensure the correctness of processing results. At the same time, it implies that all information subject to processing needs to be known to and stored by all nodes in the network. Otherwise, redundant execution would not be possible. Consequently, private or confidential data must not be processed on the blockchain — it would immediately leak into the network.

Centralized applications — the services we know and use — have intransparent and trust-based privacy guarantees [109, 189, 354]: Sensitive information is stored in databases behind company walls, which cannot be observed by users or other outside parties. After information crosses the provider's boundary, it can no longer be tracked or inspected. Users, therefore, trust the application provider to handle their data responsibly and in compliance with data protection regulations. The approach to privacy is organizational, through isolation, rather than technical.

Decentralized applications can not rely on the same mechanism of hiding data behind service boundaries. Privacy has to be addressed on a technical level. Data used in on-chain processing is and has to be available to all validating peers in the network. This renders privacy guarantees much more binary: Privacy is guaranteed or not.

Unfortunately, there are few approaches to incorporate privacy requirements into the design or implement privacy of decentralized applications today. Existing dApps mostly consider privacy a challenge to be solved in the future and hope that it will be dealt with by the blockchain platform they build on. In contrast to scalability, privacy is addressed less in research. The existing research is focused on infrastructure-level privacy with a strong emphasis on payments [252, 293]. These insights cannot directly be transferred to address the wide range of privacy challenges decentralized applications have.

Private and permissioned blockchain networks, e.g., IBM's Hyperledger Fabric [13] or JP Morgan's Quorum [79], at first glance, appear to have an advantage with regards to privacy: By isolating the network from the outside world through access restrictions, information can be hidden from external parties. Within the network, however, all information is still visible to all nodes processing transactions, and the same challenges public networks face regarding privacy and confidentiality apply [11]. While private networks may be suitable in some contexts, they do not solve the fundamental technical challenges of scalability and privacy protection. At the same time, they remove core blockchain capabilities by restricting access: External parties can no longer be convinced of transactions taking place in the network, which prevents non-duplicable digital assets as well as trustworthy references to the blockchain's history. This lack of public

verifiability significantly restricts possible use case [355].

In summary, the need for novel approaches to privacy and scalability of decentralized applications is well recognized. Yet, it is not clear how these problems can be addressed from a technical perspective. To provide further insight into and to demonstrate the significance of the multifaceted privacy and scalability requirements encountered by decentralized applications, we describe three blockchain application contexts in more detail.

1.1.1 Peer-to-peer Energy Trading

With the rise of renewable energies, the roles in the energy market change [154]: The traditional setting, where energy physically flows from few large producers to a large set of consumers, starts to cease. It is superseded by a new system of decentralized energy production. So-called prosumers not only consume but also produce energy on a small scale; for example, through solar panels. Instead of only buying from and selling to electricity suppliers as in today's market, these prosumers could profit from directly trading energy with each other [148].

As they mutually distrust each other, blockchain-based decentralized energy trading has been proposed to eliminate the need for trust [240, 246]. However, energy consumption data is highly sensitive [265, 266]. Load profiles leak precise information about consumer behavior [251, 282]. Hence, a system where this data is published to a blockchain to be processed could never be deployed in practice. Novel solutions need to be found to address these privacy concerns while still leveraging the blockchain's core properties in a decentralized trading application. Besides this privacy issue, high-volume on-chain marketplace designs are prohibitively expensive due to the blockchain's lack of scalability [50].

1.1.2 Efficient Blockchain Relays

Several blockchain networks have been successfully deployed, e.g., Bitcoin [258], Ethereum [74], and Tezos [158]. Yet, these networks largely exist in isolation. Transactions are bound to a single ledger. Smart contracts never interact with components outside the blockchain they have been deployed to. Consequently, decentralized applications are hard to migrate and can suffer from lock-in effects [342].

Different approaches to address these limitations by establishing interoperability between blockchains have been proposed [73]. Hereby, blockchain relays are particularly promising as they are the only proposal that does not introduce additional trust assumptions: Target blockchain A validates the header-chain of source blockchain B as part of its consensus and thereby enables simplified payment verification (SPV) [73, 258]. Blockchain B becomes a so-called sidechain to blockchain A and can verify A's events or states on submission [22].

Existing implementations of such relay-based sidechains, however, are prohibitively expensive. Every block of the source chain has to be submitted to and verified by the target blockchain. BTC relay [67], for example, a relay from Bitcoin to Ethereum, has not validated a block for over two years at the time of writing. Catching up with Bitcoin would require over a million Ethereum transactions of ~194,000 gas each, adding up to an overall cost of ~288,000 €.

In summary, relay-based interoperability is hindered by the validation overhead on the target ledger. Connecting several blockchains would entirely consume their processing capacity for header validation. Clearly, a more efficient and scalable way to realize blockchain relays is required.

1.1.3 Token Transfers

Blockchains, for the first time, enable non-duplicable digital goods, which are represented as virtual tokens. In Bitcoin, this property is used to create a digital currency, whereas Ethereum uses it to pay for computational tasks run on the blockchain platform. Besides such protocol-native tokens, programmable blockchain platforms allow decentralized application developers to create their own tokens through smart contracts. Such tokens frequently implement a standardized interface to ensure compatibility with exchanges and other applications, e.g., the ERC-20 standard [332]. In the context of dApps, tokens are not only used as a means of value exchange but can grant special rights to their owner, e.g., voting power or access to a service.

Like in the case of protocol-native tokens, transfers are processed on-chain and are hence visible publicly. Albeit user identity is obscured through their addresses acting as pseudonyms in such transfers, this form of privacy protection is insufficient. There is an extensive body of scientific research which shows that analysis of blockchain transaction information can reveal user identities [12, 239, 286, 291].

There exist dedicated blockchain networks for private payments, e.g., Zcash [361] or Monero [252], but their privacy protection is specific to the protocol-native token. Clearly, there is a need for privacy-preserving transfers of generic tokens employed by decentralized applications on general-purpose blockchains.

1.2 Research Questions

To clearly define the scope and direction of our research, we formulate our first problem statement by condensing the previous discussion of privacy and scalability challenges for decentralized applications:

Problem Statement 1: *Privacy and scalability are first-class requirements for decentralized applications, not retrofittable afterthoughts. At the same time, the current technological privacy and scalability options for these applications are insufficient.*

The first research question follows naturally from this problem description by asking how to advance the state of the art.

Research Question 1: *How can the fundamental privacy and scalability challenges faced by blockchain-based, decentralized applications be addressed?*

Furthermore, we formulate a working hypothesis which facilitates inquiry through the specifica-

tion of a falsifiable statement and hence sets the direction of our research.

Hypothesis 1: *Off-chaining is a mechanism to address privacy and scalability concerns of decentralized applications on a technical level.*

A confirmation of this hypothesis — and hence an answer to the research question it was derived from — would provide a direction to look in when designing decentralized applications with scalability and privacy in mind. However, it would not directly result in a technical solution that could be leveraged by developers from an engineering perspective. This motivates our second research question, which addresses the following problem:

Problem Statement 2: *Developers are ill-equipped to address the privacy and scalability needs of decentralized applications through off-chaining.*

Again, the question directly follows by asking how the status quo could be advanced:

Research Question 2: *How can developers be supported in engineering decentralized applications with better privacy and scalability properties through off-chaining?*

Based on the insights gained from studying the first research question, we formulate the following hypothesis, which we proceed to investigate in this thesis:

Hypothesis 2: *High-level programming abstractions and related tools for scalable and privacy-preserving off-chain computations can be found that enable developers to realize improved variants of decentralized applications.*

1.3 Contributions

In this thesis, we make four main contributions that address the research questions previously posed and thereby significantly advance the state of the art in scalability and privacy of blockchain-based applications. Hereby, Contributions 1.3.1 and 1.3.2 address Research Question 1, whereas Contributions 1.3.3 and 1.3.4 address Research Question 2.

1.3.1 Off-chaining Patterns for Privacy and Scalability

In response to the first research question, we introduce off-chaining as a fundamental approach to address the privacy and scalability challenges that arise in the context of blockchains, especially decentralized applications.

We define off-chaining as moving computation and/or data off the blockchain while impacting key properties as little as possible. The core idea is to minimize data storage as well as computational effort on the blockchain by employing blockchain-external resources. Reducing the on-chain processing footprint frees up capacity for other decentralized applications. Furthermore, storing sensible data off-chain is the only way to ensure privacy.

To bridge the gap between this abstract definition and practicable off-chaining approaches, we observe recurring challenges and solution ideas in decentralized application design. We structure and categorize them into five distinct off-chaining patterns:

1. Content-Addressable Off-chain Storage Pattern
2. Verifiable Off-chain Computation Pattern
3. Off-chain Signatures Pattern
4. Optimistic Finalization Pattern
5. Low Contract Footprint Pattern

Instantiations of these patterns allow developers to address scalability and privacy challenges they commonly face during the design of blockchain-based applications. While both privacy and scalability are not in conflict and can be addressed through off-chaining at the same time, we find that there is a tradeoff with availability that requires careful consideration.

1.3.2 Identification and Comparison of Off-chain Computation Approaches

Drawing from our analysis, we conclude that off-chain computations, as described in the verifiable off-chain computation pattern, are particularly powerful in addressing privacy concerns in the context of decentralized applications as they enable trustworthy processing of off-chain data. Furthermore, if on-chain verification of results computed off-chain is cheap, this approach can directly improve a blockchain's throughput. While some realizations of off-chain computations for specific contexts have been proposed in literature [83, 193, 320, 367], there exists no systematic analysis of possible approaches, their properties, and how they compare.

As our second main contribution, we address this issue by systematically exploring the design space for off-chain computations, identifying fundamental categories of off-chain computation approaches, and categorizing existing proposals from white and gray literature.

After providing definitions of off-chain computations and off-chain computation protocols, we describe the basic architectural components involved in their realizations. Based on these definitions, we explore the design space for off-chain computation protocols and identify interactivity, responsibility, and sanguinity as its three dimensions.

Building on these results, we introduce four categories of off-chain computation approaches that group protocols with similar characteristics and a common implied architecture:

1. Verifiable Off-chain Computations
2. Enclave-based Off-chain Computations
3. Secure Multiparty Computation–based Off-chain Computations
4. Incentive-driven Off-chain Computations

We discuss existing system and protocol proposals from white and gray literature for each category. Subsequently, we conduct a comparative assessment, which contrasts the approaches with respect to scalability, privacy, security, and programmability.

As a result of our studies, we learn that zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) from the group of verifiable off-chain computations are a particularly powerful approach to address scalability and privacy. Yet, zk-SNARK-based off-chain computations are difficult to instantiate; programmability and efficiency are major concerns.

1.3.3 Framework for Scalable and Privacy-preserving Off-chain Computations

The previous analysis led to the identification of zk-SNARK-based verifiable off-chain computations as a suitable off-chain computation approach, but applying them is hard. Computations need to be specified in difficult-to-use low-level abstractions, and on-chain verification is complex as it requires deep knowledge of the employed verifiable computation schemes.

To address this gap, we design, implement, and evaluate ZoKrates, the first framework for efficient zero-knowledge off-chain computations.

ZoKrates enables decentralized applications to address their privacy and scalability needs by providing a developer-friendly abstraction for the specification, off-chain execution, and on-chain verification of zk-SNARK-based verifiable off-chain computations.

It comprises a domain-specific language that captures the peculiarities of the underlying abstractions and allows developers to conveniently specify off-chain computations as higher-level programs. These programs are then translated into the custom ZoKrates intermediate representation (ZIR), and executed with the ZoKrates toolchain. Afterwards, a correctness proof for this program execution can be generated. To enable on-chain verification, ZoKrates supports the generation and export of verification smart contracts, which check the correctness of off-chain computations.

1.3.4 Privacy-preserving and Scalable Variants of Decentralized Applications

In an extensive evaluation, we demonstrate how the privacy and scalability problems faced by the applications introduced in Sections 1.1.1 to 1.1.3 can be addressed through verifiable off-chain computations.

Concretely, we discuss ZoKrates-based variants of decentralized applications for privacy-preserving token transfers and peer-to-peer energy trading as well as a scalable blockchain relay, thereby demonstrating the viability of Contribution 1.3.3.

We independently proposed, implemented, and evaluated the decentralized peer-to-peer energy trading application and the scalable blockchain relay together with our co-authors. In contrast to these efforts, the ZoKrates-based anonymous token transfer application was independently designed and built by the Ernst & Young blockchain division [132], which underlines ZoKrates' viability and maturity.

Together, these four contributions significantly advance the state of the art in privacy and scalability of blockchain-based applications. We contribute to a more precise understanding of the challenges and introduce off-chaining as a universal approach to dApp privacy and scalability. Furthermore, we provide a framework for efficient privacy-preserving off-chain computations, which was employed not only by ourselves but has seen independent use in academia and industry as a privacy and scalability engineering tool, e.g., [30, 143, 169, 175, 267, 277, 307, 314].

1.4 Published Material

Parts of this thesis are based on material that was previously published at scientific venues. In this section, we describe how and where these publications reflect in this thesis. The following list gives an overview of scientific publications by the author that contain relevant material:

[1] S. Tai, J. Eberhardt, and M. Klems. "Not ACID, not BASE, but SALT – A Transaction Processing Perspective on Blockchains". In: *Proceedings of the International Conference on Cloud Computing and Services Science*. ScitePress, 2017, pp. 755–764.

[2] J. Eberhardt and S. Tai. "On or Off the Blockchain? Insights on Off-Chaining Computation and Data". In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 3–15.

[3] J. Eberhardt and S. Tai. "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations". In: *Proceedings of the IEEE International Conference on Blockchain*. (Best Paper Award). IEEE, 2018, pp. 1084–1091.

[4] J. Eberhardt and J. Heiss. "Off-chaining Models and Approaches to Off-chain Computations". In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 2018, pp. 7–12.

[5] J. Eberhardt, M. Peise, D.-H. Kim, and S. Tai. "Privacy-Preserving Netting in Local Energy Grids". In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 2020, pp. 1–9.

[6] M. Westerkamp and J. Eberhardt. "zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays". In: *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*. IEEE, 2020, pp. 378–386.

In [1], we provide a detailed discussion of blockchain systems from a transaction processing perspective. We analyze their properties in contrast to other types of distributed systems and their role in decentralized applications, which reflects in Section 2.2.1 of this thesis. Our definition of off-chaining, as well as the off-chaining patterns introduced in Chapter 4, are based on [2]. Off-chain computations were proposed in [3] as an approach to address privacy and scalability. Different types of off-chain computations and a survey of possible instantiations thereof were presented in [4]. An earlier, less evolved version of ZoKrates was presented in [3] as an efficient and developer-friendly framework for zkSNARK-based off-chain computations. The privacy-preserving decentralized energy trading application described in Chapter 12 was published in [5]. Finally, the scalable zk-SNARK-based blockchain relay introduced in Chapter 13 was published in [6].

For a complete list of publications, refer to the List of Publications in the back matter.

1.5 Thesis Organization

This thesis is structured into five main parts. An overview of this structure is depicted in Figure 1.1.

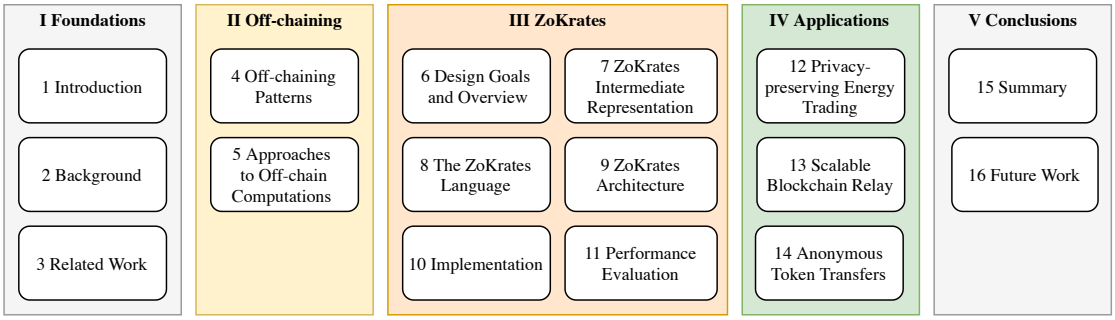


Figure 1.1: Thesis Organization

Part I, which lays the foundations of this work, comprises three chapters. After this introductory chapter, Chapter 2 provides necessary background in the field of blockchains and verifiable computations. We discuss related work in-depth in Chapter 3.

We introduce off-chaining in Part II, which comprises our first two main contributions. After giving a definition for off-chaining, we formulate and describe off-chaining patterns in Chapter 4. Studying possible instantiations of the delegated computation pattern, we identify, categorize and compare off-chain computation approaches in Chapter 5. Together, these contributions provide an answer to Research Question 1.

In Part III, we present our third main contribution: We introduce ZoKrates, our language and framework for verifiable off-chain computations and with that provide an answer to Research Question 2. In Chapter 6, we motivate the need for our solution, derive design goals, and provide an overview before theoretically deriving the ZIR as our compilation target in Chapter 7. In Chapters 8 to 10, we describe the ZoKrates language, architecture, and implementation in detail, before conducting a performance evaluation in Chapter 11.

To demonstrate viability and practicality of our approach, we provide enhanced variants of the decentralized application use cases presented in Sections 1.1.1 to 1.1.3 as our fourth main contribution in Part IV. The design, implementation, and evaluation of a peer-to-peer energy trading system are given in Chapter 12. As a second use case, we describe a scalable blockchain relay that facilitates efficient interoperability between blockchains in Chapter 13. We describe a ZoKrates-based design of privacy-preserving token transfers in Chapter 14.

Lastly, we conclude in Part V: We summarize the results of this thesis in Chapter 15 and provide an outlook in Chapter 16.

This thesis is written to be read sequentially, but Parts II and III can also be studied independently from each other. Both parts, however, differ in required background and related work. We depict how this thesis can be read under the consideration of these dependencies in Figure 1.2.

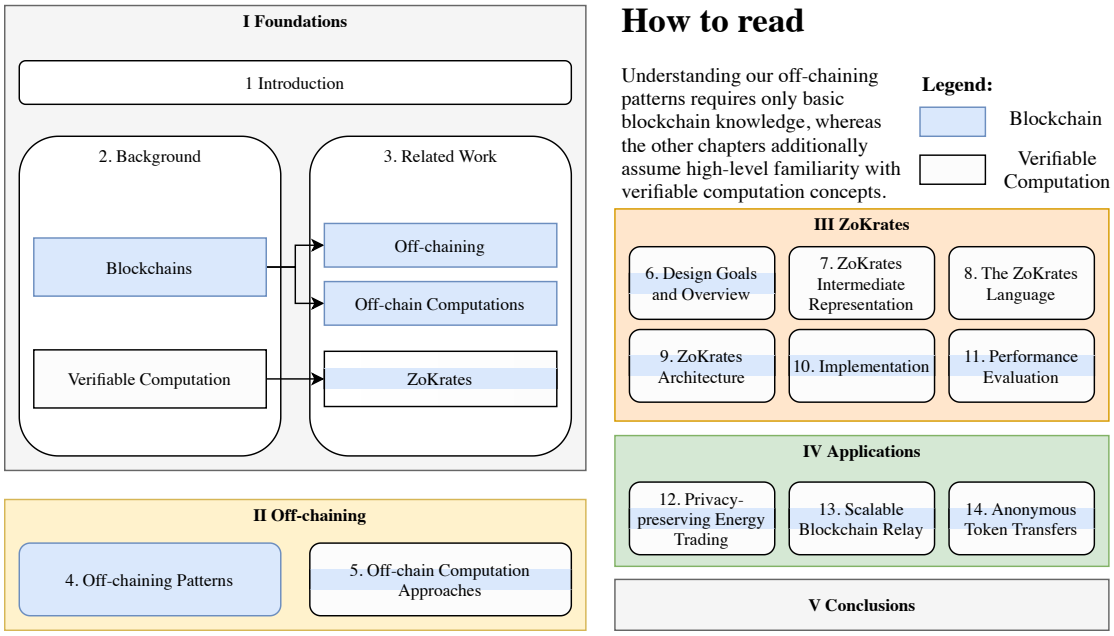


Figure 1.2: How to Read & Dependencies

CHAPTER 2

Background

In this chapter, we provide relevant background on blockchains and verifiable computation schemes as a foundation for the remainder of this thesis. After introducing the essence of blockchain technologies using the examples of Bitcoin and Ethereum, we discuss smart contracts as well as private and permissioned blockchains in more detail. Our work on off-chain computations presented in Section 4.3.2, and ZoKrates proposed in Part III in particular, heavily relies on verifiable computation schemes. We provide necessary background in the second part of this chapter.

This chapter is based on material previously published at CLOSER 2017 [319] and IEEE Blockchain 2018 [113].

2.1 Blockchains and Smart Contracts

Technically, a blockchain is a data structure that collects transactions in cryptographically linked blocks. More commonly, however, the term is used to describe distributed peer-to-peer systems that implement a trustless shared append-only transaction ledger [319]. Deployments of such systems are referred to as blockchain networks. Trustless processing describes a form of transaction processing that does not rely on a trusted third party or other intermediaries to guarantee correctness, but verification by independent peers [318].

To present the core concepts of blockchain technologies, we subsequently introduce Bitcoin and Ethereum, two systems that represent significant milestones in their development in Sections 2.1.1 to 2.1.2. Besides introducing the blockchain data structure, Bitcoin combines the Proof-of-Work (PoW) consensus algorithm with an incentive structure to economically motivate peers to contribute to the network. Expanding on these ideas, Ethereum allows the deployment of complex user-defined programs that are executed by the blockchain network, so-called smart contracts.

Applications that rely on blockchain infrastructure — and smart contracts in particular — as a

key part of their architecture in order to depend on centralized components as little as possible are referred to as decentralized applications. We provide additional background on this concept in Section 2.1.3.

Bitcoin and Ethereum are public and permissionless blockchains. Nodes have equal capabilities and access to the blockchain network is unrestricted. In contrast, a central entity explicitly manages node capabilities and network membership in permissioned blockchains. We provide definitions and examples in Section 2.1.4.

A plethora of blockchain systems have been proposed in research and industry besides Bitcoin and Ethereum. Conceptually, however, they follow similar ideas and only deviate in specific aspects that are not particularly relevant in the context of our work, e.g., consensus algorithms or governance. A comprehensive overview of existing platforms by market capitalization is given in [214].

2.1.1 Bitcoin

Bitcoin, the first blockchain-based system, was proposed in 2008 by Satoshi Nakamoto [258]. Its goal was to create a decentralized digital currency that could not be controlled by a central authority and would enable peer-to-peer value transfer without relying on a trusted intermediary. For that, the Bitcoin protocol combines transactions secured by asymmetric cryptography with a consensus algorithm to decide on the transaction order within a network. Peers independently validate transactions and thus do not need to rely on a trusted party. In addition to that, however, a global order of the transactions has to be decided on to prevent double-spending of digital funds. Again, this has to be done in a way that does not re-introduce a trusted third party.

Since traditional consensus protocols, e.g., Paxos [215] and Raft [263], do not scale as required, the PoW consensus protocol was developed as a main innovation of the Bitcoin system. This protocol allocates the right to add transactions to the network in proportion to the computational effort spent by a peer to secure the network. By linking rights to computational power instead of identities, the PoW algorithm does not only avoid the communication complexity of traditional consensus algorithms but simultaneously ensures Sybil-resistance [107]. Since directly agreeing on a transaction order is expensive, multiple transactions are grouped into a block for efficiency. These blocks are then ordered by consensus. Each block references its predecessor, which implies a chain data structure – the blockchain.

Internally, Bitcoin uses a simple and limited stack-based scripting language to define transaction validation logic. This language is called Bitcoin Script and is purposefully not Turing-complete to guarantee termination of script executions. Without this property, the networks liveness could be impaired: Nodes validating transactions could be stuck in an infinite loop, for example. Bitcoin script allows developers to define conditions for spending a bitcoin. However, the limited scope of this language makes the realization of other blockchain applications and non-payment related functionality difficult [74]. Proposals like Colored Coins [19], Namecoin [259], and the idea of autonomously executing legal contracts [317] show that implementing use cases unrelated to payments is difficult and requires forking into separate networks.

2.1.2 Ethereum and Smart Contracts

Extending Bitcoin's idea of peer-to-peer value transfer, Ethereum, a trustless computing platform, was proposed in 2014 [74, 350]. Ethereum adds a Turing-complete and stateful programming abstraction to the blockchain, which enables the execution of complex user-defined code without trusting a server or central party. Trust in a central processor is replaced by independent validation of each program execution by every peer in the network and agreeing on an outcome. The resulting system is described as a *world computer* since it can be seen as one global state machine with a user-programable state transition function.

In Ethereum, the programs that define the system's state transition function and are executed in a trustless and tamper-proof manner in the network are called smart contracts. Here, smartness does not refer to artificial intelligence, but rather to the enforcement of contract terms by an autonomous self-executing agent. The notion of smart contracts was initially introduced in the context of technical representations of legal contracts [317]. However, in the context of this thesis, we use the term smart contract more generally to refer to a set of automatically enforced digital rules which cannot be manipulated or censored as implemented in Ethereum. Consequently, smart contracts have many more use cases where automatically executed complex conditional logic is required beyond legal contracts.

Two important properties need to be ensured for on-chain computations as defined through smart contracts: determinism and termination. To ensure consistency among nodes, state transitions defined through smart contracts have to be deterministic. Otherwise, peers could disagree on the outcome of valid executions, which would make finding consensus in the network impossible. Therefore, filesystem and network access, for example, cannot be permitted, as they might yield different results when performed on different nodes in the network. Second, due to the redundant execution of smart contract logic on every node in the network, operations blocking nodes need to be forbidden to ensure liveness (e.g., infinite loops, long-running transactions). For the network to make progress and for block times to average the targeted block interval, timely termination of all processing steps has to be guaranteed; there can be no infinite loops or long-running transactions. The problem of preventing infinite loops theoretically reduces to the halting problem: Does a given program terminate or will it run forever? It is a well known standard result of theoretical computer science, that this problem is undecidable for Turing-complete languages, so another way of preventing infinite loops has to be found [327]. As explained earlier, Bitcoin solves this problem by limiting its scripting language to prevent the specification of non-terminating programs. Ethereum, in contrast, exposes a Turing-complete programming abstraction and has to ensure termination for arbitrary code. To circumvent the undecidable halting problem, Ethereum introduced the notion of gas: Every operation is assigned a cost and an initial endowment is specified for the invocation of a smart contract function. During execution, each operation consumes gas until the endowment is used up. With that, every function execution is guaranteed to halt. Even if it contained an infinite loop, the endowment would be used up at some point and cause the execution to stop. The concept of gas is further used to define an upper bound for the cost of block validation for peers in the network: A block gas limit restricts the possible load (in terms of computation and storage) and also defines an upper complexity for operations that can atomically be performed on the blockchain.

Ethereum smart contracts are executed by a dedicated virtual machine, the Ethereum Virtual Machine (EVM) [350]. This machine takes care of gas accounting during execution and its instruction set guarantees determinism by design. Few basic operations, e.g., hash functions, that are often required but costly to execute natively in the EVM are supported by so-called *precompiled contracts* [351]. They can be called like other smart contracts but have a discounted gas cost since they are directly implemented in Ethereum clients. Writing EVM code directly, however, is comparable to programming in an assembly language. Thus, to provide developers with more efficient means for developing smart contracts, higher-level programming abstractions that target the EVM have been developed, e.g., Solidity [126], Vyper [127], and LLL [124]. Solidity, for example, a statically typed smart contract programming language with JavaScript-inspired syntax provides an object-oriented programming model [126]. We use Solidity throughout this thesis to represent smart contracts as it represents the most mature and established smart programming language targeting the EVM [160].

2.1.3 Decentralized Applications

Modern applications are commonly highly distributed and comprise many independent components and services [139]. They rely on servers to execute business logic, on databases and storage systems for persistence, and on messaging services for communication. These systems can be self-hosted by the application provider or purchased on-demand from a cloud provider, for example. Albeit at slightly different degrees, the application provider maintains control over the systems in both cases. Control is centralized and users' trust in the application provider is essential.

As a counterpoint to this well-established category of applications, the term decentralized application (dApp) was coined to refer to applications that do not rely on centralized infrastructure. Instead, they leverage decentralized systems to realize applications that are not controlled by a single entity, resistant to censorship, as well as highly available and fault-tolerant. The term dApp is particularly prominent in the context of blockchain-based applications where smart contracts that specify business logic are combined with off-chain components, e.g., client-side user interfaces and storage [74, 284]. Once deployed, smart contracts are executed by the corresponding blockchain network in a censorship-resistant, transparent, and verifiably correct way. The role of the application provider is reduced to the role of the application developer unless privileges are programmed explicitly. Besides typical end user–facing dApps, there are also dApps that solve problems on the infrastructure level, e.g., realize oracles [173] or blockchain relays [73]. An overview of existing dApps can be obtained through online repositories that collect dApps for different blockchains and facilitate their exploration [96, 97, 313]. For statistics on usage and underlying blockchain networks, refer to [281, 353].

For the purpose of this thesis, we follow the definition above and use the term dApps to describe applications built with smart contracts as the components implementing key business logic. It should be noted, however, that the idea of dApps can be interpreted more generically: Decentralized storage systems (for example, IPFS [40], Filecoin [278], or Swarm [324]) or peer-to-peer messaging systems (e.g., Whisper [131]) represent decentralized solutions for common tasks in application design and can be used to implement applications that are not controlled by a single

entity without relying on smart contracts.

2.1.4 Private and Permissioned Blockchains

The Bitcoin and Ethereum networks are public. Arbitrary new nodes can join the network at any time. Additionally, all nodes in the network have equal rights.

This is not the case for all blockchain systems, however. To cater to the needs of enterprises, Hyperledger Fabric [13], R3's Corda [66], or Quorum [79] deliberately limit network access, introduce the concept of identities, and explicitly define permissions for nodes. While public verifiability is no longer given in these networks, other desirable properties like confidentiality, scalability, and finality are often easier to achieve [333].

There is no generally agreed-on way to distinguish between different types of blockchain designs and corresponding deployments. Subsequently, we introduce the two most established perspectives.

One way to categorize different designs is by node permission. Although this distinction is commonly made in literature, the specific descriptions differ and are inconsistent [13, 333, 355]. Consequently, we provide our own definitions for a permission-based categorization, drawing inspiration from the referenced work: Both read and write access to a blockchain can be restricted. Hereby, read access (generally) coincides with membership in the blockchain network. Consequently, we refer to blockchain networks that can be joined and read by arbitrary nodes as public. Networks that restrict read access are called private. Blockchain networks in which all nodes have equal capabilities to create blocks and thus write to the blockchain are called permissionless. Networks, in which only a subset of nodes is allowed to write to the blockchain, are called permissioned. Table 2.1 summarizes this categorization of blockchain technologies and deployments.

Table 2.1: Categorization of Blockchain Deployments by Permission including Examples

Reading \ Writing	Permissionless	Permissioned
Public	Bitcoin [258], Ethereum [74]	Corda [66]
Private	Ethereum on Azure [121]	Hyperledger [13], Quorum [79],

A second way is to distinguish between different blockchain deployments is by control [75]. This distinction does not consider read access but categorizes by the ability of real-world entities to write to the blockchain. In *public* blockchain networks, arbitrary nodes are allowed to write new blocks. Blockchain networks in which this ability is constrained to a set of real-world entities are referred to as *consortium* blockchains. Only in the extreme case, where a single entity fully controls write access, the blockchain is called *private*.

In the context of this thesis, we generally follow the permission-oriented terminology unless stated otherwise.

2.2 Verifiable Computation

The research area of verifiable computations develops and analyzes protocols that allow a client to outsource a computation to a worker in a way that it can check the correctness of the computation's result computed by the worker on the client's behalf. The worker does not have to be trusted. Here, the worker acts as a prover and the client as a verifier. The verification of a delegated computation has to be cheaper than re-execution for the client. Otherwise, there would be no advantage over client-side execution in the first place.

More formally, verifiable computation is concerned with the following problem [64]: A verifier provides a program C with inputs $\vec{x}$ to a prover. It then receives a result claim $\vec{y}$ from the prover. How can the verifier V be sure that the prover P correctly computed $\vec{y} = C(\vec{x})$, the result of a program C for inputs $\vec{x}$? To solve this problem, V engages P in a protocol that allows V to ensure that P computed C correctly. Generalizing [150], we call such protocols *verifiable computation schemes*. In the theoretical community, which is less concerned with the application of proofs in the context of verifiable computations but reasons about efficient proofs for arbitrary statements in general, such two-party protocols between V and P are commonly referred to as interactive proof systems [156].

Research in verifiable computation is oftentimes motivated in the context of cloud computing use cases [336]. Instead of having to trust cloud providers, clients could independently and cheaply verify execution correctness by adopting verifiable computation schemes. In the context of this thesis, however, we use verifiable computation schemes to allow a blockchain to delegate computations to blockchain-external resources and cheaply verify their correctness. We introduce this idea in detail in Part II.

For a long time, this area of research was purely theoretical. The proposed verifiable computation schemes based on probabilistically checkable proofs (PCPs) [39, 103] or fully homomorphic encryption (FHE) [86, 150] were impractical due to immense proof sizes, high asymptotic complexity, and large constants. In recent years, however, advancements in complexity theory and cryptography led to the development of novel and improved verifiable computations schemes that can be considered practical depending on the use case [336].

In this section, we provide basic definitions, discuss important properties, and provide an overview of verifiable computation schemes that are relevant throughout the later parts of this thesis. We start with a description of key properties of verifiable computation schemes and discuss which computations can be proven. Afterward, we provide an overview of state-of-the-art theoretical verifiable computation schemes and compare their properties. In the process, we introduce the notion of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) in more detail, as it plays a particularly significant role in later parts of this thesis.

2.2.1 Foundations of Verifiable Computation Schemes

In this section, we define verifiable computation schemes and describe which computational statements they can prove. Afterward, we briefly discuss properties that characterize verifiable computation schemes and are relevant in the context of this work.

Definition:

We previously defined verifiable computation schemes as two-party protocols between a verifier V and a prover P that allow V to check whether P computed the result $\vec{y}$ of a program C correctly, i.e., $\vec{y} = C(\vec{x})$. Such protocols must satisfy the following properties [185]:

- **Completeness:** P must be able to convince V of correct results. If P follows the protocol rules and $\vec{y} = C(\vec{x})$, V always accepts.
- **Soundness:** P cannot convince V of incorrect results. If $\vec{y} \neq C(\vec{x})$, V always rejects except with negligible probability.

The soundness definition above describes statistical *information-theoretic soundness*. Even an infinitely powerful prover cannot convince V of an incorrect result (except with extremely little probability). Assuming a computationally bounded prover, this strong soundness definition can be relaxed to the weaker notion of *computational soundness*: A computationally bounded P cannot convince V of incorrect results except with negligible probability. Conversely, an indefinitely powerful P could convince V of false results. To distinguish them from information-theoretical sound interactive proof systems, computationally sound interactive protocols are referred to as *arguments* [62, 200]. The relaxation of the soundness requirement led to the development of significantly more efficiency schemes [244] and made arguments the state of the art in verifiable computations [151, 164]. Note that we made no assumptions about P's behavior in either case. P can behave maliciously.

Provable Computations:

State-of-the-art verifiable computation schemes (see Section 2.2.2) allow proving arbitrary $\mathbf{NP}$ statements [17], but rely on specific characterizations of $\mathbf{NP}$ to achieve efficiency. Hence, to be provable, a computation C has to be formulated as an instance of an $\mathbf{NP}$-complete problem, e.g., as a circuit satisfiability or constraint satisfaction problem. Unfortunately, in practice, programs are not in such form.

Useful programs are commonly represented in high-level programming language such as Java [16] or C [196]. By design, programs in such languages are representable in the random access machine (RAM) model of computation [89]. The RAM model is equivalent to the Turing-machine model of computation in power. A standard result in complexity theory states that a Turing-machine with a bounded number of computational steps and fixed input size can be unrolled into a polynomial-size boolean circuit [25]. Thus, theoretically, any high-level language program with a fixed number of inputs that halts after a finite number of steps can thus be represented as a circuit satisfiability problem. This circuit satisfiability problem can then be transformed to any characterization of $\mathbf{NP}$ used by a verifiable computation scheme through polynomial-time

reductions. More concisely, any halting program with a fixed input size can be represented as an $\mathbf{NP}$ statement.

We explain how computations can be efficiently described as $\mathbf{NP}$ statements for characterizations of $\mathbf{NP}$ commonly used in verifiable computation schemes, namely boolean circuits, arithmetic circuits, and rank-1 constraint systems (R1CSs), in detail in Chapter 7.

Characterizing Properties:

Subsequently, we describe properties that are relevant in the context of this work and characterize verifiable computation schemes.

(Non-)Interactivity: Goldwasser et al. [157] and Babai [21] independently proposed interactive proof systems where a computationally unbounded prover convinces a computationally bounded verifier in multiple rounds of interaction. In both protocols, the verifier uses random coins [25] to select challenges. Since messages in applications of verifiable computation protocols are frequently exchanged over networks, a low communication complexity is desirable. Fiat and Shamir [133] discovered a heuristic that allows the conversion of public-coin interactive protocols to non-interactive versions. Instead of being randomly selected by a verifier, challenges are selected by using a cryptographic hash function, which acts as a proxy for a random oracle. Computationally sound state-of-the-art verifiable computation schemes (see Section 2.2.2) can convince a verifier in one message, i.e., they are non-interactive.

Zero-Knowledge: A zero-knowledge proof of a statement reveals nothing but the correctness of the statement. Goldreich et al. proved the existence of zero-knowledge proof systems for arbitrary $\mathbf{NP}$ statements by developing a zero-knowledge interactive proof for the $\mathbf{NP}$-complete graph 3-colorability problem [156]. In the context of verifiable computations schemes, the zero-knowledge property allows a prover to use private information in a computation since the proof reveals nothing but the fact that the computation was executed correctly. The prover's witness remains secret. Interestingly, the zero-knowledge property is generally satisfied by efficient verifiable computation schemes since the goal of hiding prover information and achieving small proof sizes are naturally aligned.

Succinctness and Fast Verification: Efficiency is a key requirement that needs to be satisfied by verifiable computation schemes designed to be used in practical applications. One important aspect of efficiency is communication complexity, i.e., the amount of information that has to be transmitted to a verifier by the prover. In non-interactive protocols, communication complexity reduces to proof size. As proof for an $\mathbf{NP}$ statement, a witness w can be considered a baseline for communication complexity. Proof systems with sublinear communication complexity are called *succinct* [153]. Formally, for an instance x of an $\mathbf{NP}$ problem and witness w, a proof system is succinct if its communication complexity is bounded by $o(|x| + |w|)$. It is considered unlikely that information-theoretically sound interactive proof systems can be succinct [153]. By relaxing this assumption to computational soundness, however, sublinear communication complexity can be achieved for interactive [200] as well as non-interactive proof systems [244]. Besides

communication complexity, another important aspect with regards to the efficiency of verifiable computation schemes is the verifier's running time. Sometimes, proof systems with sublinear verification complexity, i.e., running times bounded by $o(|x| + |w|)$, are also called succinct [46]. However, following [151], we refer to this property as *fast verification*. Note that verification of a computation can only ever be as fast as reading the computation's description. Without knowing the computation, the verifier cannot perform meaningful checks. Thus, preprocessing is required to compress a circuit's representation to achieve fast verification (unless the computation has some special structure like repeated sub-computations) [85].

Knowledge-Soundness: As previously explained, there exist zero-knowledge proof systems for arbitrary $\mathbf{NP}$ statements [156]. Let L be an $\mathbf{NP}$ language, V_L a polynomial-time verification algorithm, and x an instance. Generally, a zero-knowledge proof of $x \in L$ is an existential statement. It proves that there exists a w for a given x so that $V_L(x, w) = 1$. However, the proof itself does not prove that the prover knows a witness w. A *proof of knowledge*, in contrast, convinces the verifier that the prover actually knows a witness w for x. A formalization of this concept using a knowledge extractor is given in [151]. Verifiable computation schemes with proofs of knowledge are *knowledge-sound*. In the context of verifiable computations, relying on proofs of knowledge ensures that a prover correctly executed a program C and used only inputs actually available to it, not only that it is possible to do so given the required information.

Preprocessing: Some verifiable computation schemes leverage a one-time preprocessing step which generates information that is used in later protocol stages to improve efficiency. This preprocessing step is also referred to as a *setup*. Some verifiable computation schemes require information to be derived from secret randomness in the preprocessing step. Sampled random values must not be published and have to be discarded on completion of the setup for the scheme to be secure. Such a preprocessing procedure is called a *trusted setup* since only the party that performed the setup can be sure that it discarded the information and all others have to trust. The information generated in a trusted setup is commonly referred to as common reference string (CRS) [52, 94]. Depending on trust assumptions, the CRS can be generated by a designated verifier, by a trusted third party, or in a secure multi-party computation (MPC) so that one honest participant suffices to guarantee soundness [36]. A scheme without a trusted setup is referred to as *transparent*. Note that transparency does not imply the absence of a (trustless) setup. A preprocessing verifiable computation scheme that requires a setup to be performed for every $\mathbf{NP}$ statement is *computation-specific*. The generated CRS depends on the specific statement. In contrast, schemes with *universal* setups can reuse the results for arbitrary statements up to a size bound. Schemes with *updatable* CRS allow arbitrary participants to add their own (secret) randomness to the CRS at any point in time [165].

Public Verifiablility: In a publicly verifiable proof system, any verifier can verify the prover's claims using only public information [151]. This is not the case in a designated verifier system.

2.2.2 Overview of Verifiable Computation Schemes

Subsequently, we provide a brief overview of state-of-the-art verifiable computation schemes. Since we strictly focus on general computations in the context of this thesis, we only consider schemes for arbitrary $\mathbf{NP}$ statements and do not discuss proof systems for specific statements, e.g., Schnorr signatures [296] or Groth-Sahai proofs [167].

We want provers to be able to use private information in their computations. Hence, we only consider zero-knowledge schemes. Most, but not all of the presented verifiable schemes are zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs), a term often used in literature to describe particularly efficient schemes that have fast verification in addition to their eponymous properties.[1] While all schemes produce non-interactive and knowledge-sound arguments, succinctness is not generally achieved.

When comparing schemes for practical applications, concrete efficiency is at least as important as asymptotic behavior. Thus, we only include schemes that have been evaluated with regards to concrete efficiency and ideally have been implemented. We structure our overview of schemes by the type of computation they are most suitable for as proposed by [231].

One-time Execution of Large Computations

In a setting where the correctness of a program's execution should be proved only once, a verifiable computation scheme cannot benefit from a (costly) program-specific setup that would amortize over future verifications. For large programs, asymptotic efficiency becomes more important than small constants. Prover and verifier running times as well as proof sizes should grow as little as possible with the complexity of the program.[2]

Zero-knowledge scalable transparent arguments of knowledge (Zk-STARKs) [33] realize such a scheme: Prover running time is quasilinear in the complexity of the program, verification is polylogarithmic, and proofs are of logarithmic size. The scheme relies solely on public randomness, i.e., is transparent, and does not require a trusted setup. However, concrete efficiency is comparatively low. Ligero [9], another transparent scheme with quasilinear prover runtime, has significantly better concrete efficiency. Verifier runtime and proof size, however, grow with the square root of program complexity. Aurora [37] has similar properties to Ligero with slightly lower concrete proving and verification efficiency but achieves polylogarithmic proof size.

Small Computations

For programs with low complexity, high concrete efficiency is paramount and asymptotic efficiency is less relevant. Sublinear verifier running time can be given up in favor of small constants and schemes with small concrete proof sizes for small computations dominate those with better

[1] There is no uniform definition of zk-SNARKs in the literature. In addition to the eponymous properties, fast verification and quasilinear prover runtime are often included. Some definitions tighten our sublinearity requirement for proof sizes and verifier runtime and demand polylogarithmic asymptotic behavior.

[2] Here, a program's complexity refers to the size of its representing $\mathbf{NP}$ statement.

asymptotic behavior.[3] The schemes in this category are transparent since the disadvantages of having to run a trusted setup outweigh the performance benefits for small programs.

Bulletproofs [69], initially proposed for range proofs in the context of confidential Bitcoin transactions, have a linear prover, a linear verifier, and logarithmic proof size. In particular, they have good concrete efficiency with proof sizes in the order of kilobytes, the smallest of the schemes in this category. Hyrax [335] improves on Bulletproofs' verification efficiency concretely and asymptotically. The prover is linear and verifier runtime as well as proof size grow with the square root of program complexity. However, the concrete proof sizes are the largest of the schemes presented in this category. Spartan [301] improves on Hyrax's performance and has the best concrete efficiency of proving in this category.[4] Proofs are small, comparable to those of Hyrax, but larger than Bulletproofs. Asymptotically, like in the case of Hyrax, the prover is linear and verification as well as proof size grows with the square root of program complexity. Lakonia [302] represents another alternative to Bulletproofs with slightly larger proof size but significantly faster verifier and prover. Asymptotically, the prover is linear, the verifier runtime grows with the square root of program complexity and proofs are of logarithmic size.

Repeated Execution of Large Computations

A verifiable computation scheme for computations that are executed and proven correct repeatedly can benefit from introducing an expensive preprocessing step that improves the efficiency of provers and verifiers. The cost for that one-time setup is amortized over future protocol executions. While the concretely efficient schemes presented in the previous section were sufficient and in some cases superior to preprocessing schemes for small statements, they generally suffer from weak asymptotic efficiency for more complex computations. In this section, we give an overview of verifiable computation schemes that employ a preprocessing step to achieve both good concrete and asymptotic efficiency. We group these schemes by the properties of their preprocessing step. Since all schemes achieve succinctness, we refer to them as preprocessing zk-SNARKs.

zk-SNARKs with Trusted Setup: Verifiable computation schemes that rely on a trusted setup are the most efficient schemes available today: They have constant-size proofs consisting of few group elements, usually points on an elliptic curve, and allow verifier runtimes independent of the computation to be verified. Provers, however, are generally not cheap, and trusted setups can limit applicability in settings without a trusted third party or with a dynamic set of participants that make MPC-based setups [36, 58, 59] impractical.

Gennaro et al. [151] presented a breakthrough zk-SNARK construction for quadratic arithmetic programs (QAPs) that leverages pairing-based cryptography to achieve small constant-size proofs and verification complexity that is computation-independent and only grows with the size of the statement ($|x|$). The prover has quasilinear running time. The scheme relies on a computation-specific trusted setup that generates a CRS with size proportional to the statement and witness

[3] Results with regards to concrete efficiency presented in our overview are based on a comparison of $\mathbf{NP}$ statements represented as R1CS of size 2^{20} conducted in [302].

[4] Spartan actually refers to a family of proof systems. In this section, we refer to SpartanSNARK which has sublinear verification complexity [301].

($|x|+|w|$).[5] The construction was reformulated for arithmetic circuits [268] as well as R1CS [35] and improved with regards to efficiency [95, 224], and generalized [47], eventually culminating in a construction by Groth that represents the most efficient pairing-based verifiable computation scheme available at the time of writing. Groth's construction [164] has proofs that consist of only three group elements ($<200\,\mathrm{B}$ for standard instantiations) and the verifier has to check a single pairing equation using three pairings.

In the previously introduced schemes, proofs intercepted by a third party can be re-randomized, a property referred to as *malleability*. In a malleable proof system, a proof of a statement can not only be derived from a witness, but also from valid proofs. This property is not necessarily problematic in the context of verifiable computations but has to be taken into consideration [80]. Groth and Maller [166] provide a non-malleable construction that does not increase proof sizes and adds only little verification overhead compared to [164]. To address the comparatively high prover cost, distributed proof generation has been proposed [352].

Universal and Updatable zk-SNARKs: In this section, we introduce verifiable computation schemes using pairing-based zk-SNARK constructions that still require a trusted setup to derive a CRS, but this CRS is universal, i.e., can be used for any computation.[6] Additionally, the schemes allow potential verifiers who do not trust the setup procedure can update the CRS with their own randomness. Together, these properties significantly simplify practical deployments. Computation-specific CRS are deterministically derived from the universal CRS and can thus be independently checked by verifiers without requiring trust.

The first universal and updatable zk-SNARK scheme was proposed by Groth et al. [165] and has constant-size proofs and constant-time verification. Yet, the scheme is not practical: the CRS has a size that is quadratic in the computation's complexity and its updates are expensive.

To address this issue and improve practicality, three novel verifiable computation schemes with universal and updatable CRS have emerged: Sonic [232] was the first scheme to achieve linear size CRS.[7] It is designed for arithmetic circuits (see Chapter 7) and, like previous pairing-based constructions, has constant-size proofs ($\sim 10\times$ the size of [164]), constant verifier runtime, and quasilinear provers. While asymptotically similar, proof generation has significant concrete overhead compared to the previously introduced zk-SNARKs with computation-specific trusted setups. Marlin [84] and Plonk [144] are two concurrent proposals that improve on Sonic's concrete efficiency. Both constructions achieve similar performance and reduce Sonics proving cost by an order of magnitude (to the levels of [164]) while accelerating verification by $\sim$3-5$\times$ and maintaining proof size. An in-depth comparison of the schemes' concrete efficiency is difficult since Marlin is formulated for R1CS, whereas Plonk relies on a more specific restricted arithmetic circuit. These choices lead to varying relative concrete efficiency depending on the specific

[5] The CRS contains public information party of which have to be accessed by both provers and verifiers. In implementations, these parts are commonly separated and referred to as proving and verification keys.

[6] More precisely, the CRS can be used for any computation that has an $\mathbf{NP}$ representation smaller than a predefined size bound.

[7] We refer to the *unhelped* Sonic construction, here. For batch verification of proofs, Sonic can benefit from untrusted helpers that reduce proof size and prover cost.

computation at hand.[8]

Libra [356], a verifiable computation scheme with universal and updatable CRS, is especially useful for very large computations or settings where prover memory is a bottleneck and distributed techniques are not applicable due to its linear and concretely efficient prover. However, it has logarithmic verifier runtime and proof size (and concretely larger proofs than the previous schemes) which makes it an inferior option in most cases.

Transparent zk-SNARKs: The last category of preprocessing zk-SNARKs is formed by transparent verifiable computation schemes, i.e., constructions that do not require a trusted setup, but leverage a trustless preprocessing step.[9] For the state-of-the-art schemes presented here, transparency comes at the price of worse asymptotic and concrete efficiency compared to the previous categories.

Supersonic [70] is a transparent zk-SNARK formulated for arithmetic circuits. It offers logarithmic verification time, has logarithmic proof sizes, and quasilinear prover times. Proofs are relatively small (10s of kB) and verification is fast (100s of ms), but concrete prover overhead is significant. Fractal [85], a scheme for R1CS, has more efficient provers, although their asymptotic runtime is also quasilinear. Verifier runtime and proof size are polylogarithmic. With few MBs, proofs are comparatively large. Xiphos [302] also builds on the R1CS abstraction and, like Supersonic, achieves logarithmic verifier runtime and proof size. Additionally, it achieves linear proving times and thus improves on Supersonic's prover efficiency while maintaining comparable concrete proof sizes.

[8] Plonk is particularly efficient for multiplication-heavy arithmetic circuits, whereas Marlin is more efficient in case of addition-heavy arithmetic circuits with large fan-in addition gates [144].

[9] Here, we only consider zk-SNARKs with fast verification. Other less asymptotically efficient transparent verifiable computation schemes have been presented in the section covering constructions for small computations.

CHAPTER 3

Related Work

In this thesis, we propose off-chaining as a strategy to address scalability and privacy limitations of blockchain-based applications. We make four main contributions in this context that advance the state of the art conceptually and practically. In this chapter, we discuss work related to these contributions.

We start with a discussion of work related to off-chaining in general and off-chaining patterns and computations in particular in Section 3.1, which relates to the two contributions described in Part II. In Section 3.2, we discuss work related to ZoKrates as introduced in Part III. Part IV describes three applications of ZoKrates to address scalability or privacy challenges in blockchain-based applications. Since related work is specific to each of these applications, we discuss it in the respective chapters.

This chapter reuses material previously published in IEEE Blockchain 2018 [113].

3.1 Off-chaining

We define off-chaining as the process of moving computation or data off the blockchain — to blockchain-external resources — without losing the blockchain's desirable properties (see Section 4.1).

Poon and Thaddeus introduced the related, but more narrow notion of off-chain payments in the context of payment channels [276]. Gudgeon et al. generalize this notion to off-chain transactions, which they define as the private exchange of authenticated transactions in the context of payment and state channels [168].

In reference to our work [112], Molina-Jiminez et al. introduce the concept of hybrid smart contracts that rely on on- and off-chain components during execution [249, 250]. Similarly, Li et al. propose hybrid smart contracts that leverage off-chain consortia to execute specific functions of Ethereum smart contracts off the blockchain [221].

Describing the opposite of off-chaining, Heiss et al. coined the term *on-chaining* that describes moving data onto the blockchain in a trustworthy manner and is closely related to oracle problems [173].

3.1.1 Off-chaining Patterns

Popularized in software engineering by the Gang-of-Four [146], a design pattern describes general and reusable solution templates to commonly recurring complex problems. Several design patterns have been proposed for different concerns in the context of blockchains and smart contracts. Our work is the first to introduce patterns related to off-chaining. The proposals subsequently introduced recognize and build on our results, sometimes using different names.

Xu et al. propose a blockchain pattern collection that covers several design challenges in the context of architectures using blockchains, blockchain-based applications, and smart contracts [357]. The authors distinguish data management, oracle, and security patterns as well as structural patterns for smart contracts. Wöhrer and Zdun introduce security-related patterns for Ethereum smart contracts written in Solidity [349]. In [348], they extend their work to other categories of smart contract design challenges — namely action & control, authorization, lifecycle, and maintenance. Mühlberger et al. characterize oracles by their interactions, describe these as patterns, and compare implementations quantitatively [254]. Zhang et al. apply design patterns from [146] to address domain-specific challenges faced by blockchain-based applications in the healthcare context [365]. Stengele and Hartenstein introduce the *atomic information disclosure* pattern and provide a realization using threshold encyption [315]. The pattern guarantees that results that were redundantly and independently computed off-chain are published to a blockchain atomically. This atomicity property prevents the lazy copying of previously published computation results.

3.1.2 Off-chain Computations

As we describe in detail in Chapter 5, there is a considerable amount of work on protocols and systems that realize off-chain computations leveraging verifiable computations [57, 210, 293, 314, 361], secure multi-party computations [14, 41, 213, 367], incentive-driven designs [193, 320], and trusted execution environments [83]. We provide the first systematization of the general design space and comparative analysis of these proposals.

3.2 zk-SNARK-based Off-chain Computations

With ZoKrates, we introduce the first holistic and practical framework for zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK)-based off-chain computations. In this section, we present and discuss related work that applies zk-SNARK-based off-chain computations in the context of blockchains in general and private smart contracts in particular. Furthermore, we discuss existing programming abstractions for zk-SNARKs and contrast them with the ZoKrates language.

3.2.1 Applications of zk-SNARK-based Off-chain Computations

There is a considerable amount of loosely related work that uses verifiable computation schemes, and zk-SNARKs in particular, in the context of specific blockchain-based applications.

Inspired by Zerocoin [245], Zerocash [293] leverages zk-SNARKs to realize an anonymous payment scheme which hides a payment's sender, receiver, and transferred amount. This line of academic work led to the creation of Zcash [361], a privacy-preserving cryptocurrency. zk-SNARKs have also been proposed in the context of mixers that enable privacy protection for blockchains without inbuilt guarantees [218, 300]. As shown in Part IV, ZoKrates can and has been used to implement such protocols.

Other applications of zk-SNARKs in the context of blockchains put a focus on scaling: In ZK-Rollups [123], a proposal from the Ethereum community, large sets of cryptocurrency transfers are batched and processed off-chain by special relayer nodes. These relayers process the batches and leverage zk-SNARKs to prove the correctness of the off-chain processing to the blockchain. Similarly, zk-SNARKs have been used in commit chains [197] to realize provably correct processing through sidechain operators. Coda [53] represents a blockchain design that relies on recursive composition of zk-SNARKs to compress verification of all past transactions into a single proof. By reducing the linear complexity of standard blockchain verification to checking one proof, this design enables efficient trustless synchronization for resource-constrained clients. Again, ZoKrates can serve as a tool to support the development of such protocols and systems.

3.2.2 zk-SNARK-based Frameworks for Private Smart Contracts

There exist multiple frameworks that employ zk-SNARK-based off-chain computations to realize the vision of privacy-preserving smart contracts. Unlike work presented in this section, ZoKrates does not attempt to combine specifications of on- and off-chain computations in one smart contract programming abstraction, but provides a dedicated framework for off-chain computations that can serve as a building block in such systems.

Kosba et al. introduce Hawk [210], a framework for privacy-preserving smart contracts. A Hawk program consists of a public and a private portion. This program is translated into a smart contract executed on the blockchain for the public portion, a private zk-SNARK-based off-chain computation run by dedicated manager nodes, and client-side programs that interact with these components. Manager nodes have to be trusted with regarding privacy but not correctness or availability[1]. The Hawk protocols rely on the existence of a blockchain that natively supports private payments, e.g., Zcash [361]: Untraceable coins are frozen in a smart contract before a new distribution of these coins is determined in an off-chain computation and then applied on-chain through anonymous payments. Unfortunately, few blockchain platforms support such inbuilt private transactions at the time of writing. Off-chain computations used in the evaluation of the system were specified as partially hand-optimized custom arithmetic circuits.

The zkay language [314] proposed by Steffen et al. extends the Solidity smart contract language

[1] The availability guarantee is an economic one: Managers can be financially penalized in case of protocol abortion by paying out a deposit to clients.

with privacy types that can be used in annotations to control the accessibility of private variable values. Annotated contracts can be automatically transformed into a combination of standard Solidity contracts and off-chain computation modules. Private values are stored on-chain in encrypted form and modified in zk-SNARK-based off-chain computations in a provably correct way. Unlike Hawk, the zkay system does not trust manager nodes with regards to privacy but assumes that contract participants execute off-chain computations and generate proofs themselves. The implementation of the zkay system uses ZoKrates for the specification of off-chain programs and the generation of Solidity verification contracts.

With ZEXE [57], Bowe et al. generalize Zerocash [293] to realize a blockchain-based private transaction processing system. Instead of checking conditions on coins, arbitrary predicates for so-called records are checked in zero-knowledge. Conceptually, predicates behave similarly to Bitcoin scripts in that they define conditions on transaction in- and outputs. Unlike Hawk and zkay, ZEXE hides which computation was called in a transaction, resulting in an even stronger privacy guarantee. Satisfaction of a record's pre-and post-conditions, formulated in birth and death predicates, is proven using zk-SNARK-based off-chain computations. Exemplary realizations of decentralized applications in the context of user-defined assets, e.g., private stablecoins and decentralized exchanges, are demonstrated. Yet, the transformations of generic smart contracts to an equivalent representation in ZEXE's record-based programming model remains an open challenge.

Hawk, zkay, and ZEXE make contributions in the context of the larger vision of incorporating strong privacy guarantees in general-purpose smart contract platforms. In contrast, ZoKrates is focused on making verifiable computations in the context of blockchains available to non-specialist developers and providing a privacy and scalability engineering tool for decentralized applications (dApps). As demonstrated by zkay, however, ZoKrates can be used in the implementation of proposals and provides many necessary primitives out of the box. Furthermore, it simplifies the adoption of new verifiable computation schemes in the context of higher-level protocols by encapsulating associated complexity and providing a uniform interface.

3.2.3 Specification of Verifiable Off-chain Computations

ZoKrates is the first language and toolbox for off-chain computations that supports developers throughout the full process from program specification to on-chain verification. In this section, we describe work specifically related to the ZoKrates language, our domain-specific language (DSL) for the specification of off-chain computations, and the process of translating programs to provable computational statements expressed in suitable $\mathbf{NP}$ charaterizations, e.g., arithmetic circuits or rank-1 constraint system (R1CS) (see Section 2.2.2). There are several software tools that perform a similar task, but build on different input abstractions. We choose to design a language optimized for translation into the target abstraction and offer only low-cost abstractions to the developer instead of supporting a subset of another language with unexpected limitations or language constructs without efficient translations. We discuss our design choices in detail in Chapter 8.

MPC Compilers

There is a significant body of work on how to compile higher level languages to boolean or arithmetic circuits in the context of secure multi-party computation (MPC) [72, 211, 226, 230, 283, 311]. Hastings et al. provide a survey of the state of the art of general-purpose compilers for secure multiparty computation compilers in [172]. While this line of work seems related, at first sight, there is a fundamental difference, which limits the applicability of MPC compilers for our context: Whereas the circuits in the context of secure multi-party computations are designed to actively execute a computation, and with that compute all intermediate values inside the circuit, the circuits in the context of non-interactive zero-knowledge proofs only need to assert that a given set of intermediate values satisfies the circuit. Consequently, some computations can happen outside of the circuit, which only has to check constraints on the externally computed values — a technique we leverage heavily in ZoKrates to ensure efficiency. We describe this idea in more detail in Chapter 7.

Circuit Composition Libraries

Libsnark [298], a library that implements various zk-SNARK schemes, also provides a low-level C++ abstraction that supports the specification of R1CSs by hand. To simplify this process for users, libsnark includes a collection of pre-defined low-level circuits called *gadgets* for basic operations. Bellman [54], a zk-SNARK library developed by Zcash, also provides a low-level programming abstraction for R1CS embedded in Rust [203]. Circom [180] is a library designed to support the direct specification of arithmetic circuits in a JavaScript/C-style syntax. It also provides a library of gadgets, which can be used to compose more complex circuits. Unlike ZoKrates' high-level DSL, it operates directly on the arithmetic circuit abstraction and provides syntactic elements to specify and connect those. Its circuit-oriented programming style resembles hardware specification languages like VDHL [182] or Verilog [323]. The jsnark [209] library allows the specification of arithmetic circuits in Java. Initially developed in the context of the ZEXE research paper [57] described in Section 3.2.2, the ZEXE library [299] also provides low-level programming abstractions for R1CS and a gadget library in Rust. Existing gadgets or circuits specified in one of these circuit composition libraries can be imported to and used with ZoKrates as the underlying abstractions are fully compatible.

High-Level Language Compilers

Pinocchio [268] presents a compiler that translates a subset of C to arithmetic circuits. It requires a specific source code layout and annotations. Array indices and loop bounds must be known at compile time. Extending this work, Pantry [64] introduces support for computations with state and proposes a memory abstraction based on Merkle trees. Buffet [334] introduces a more efficient memory abstraction based on permutation networks and adds data-dependent control flow to support a larger subset of C. Gepetto [91] supports the decomposition of large circuits into multiple smaller ones [91]. ZoKrates adopts techniques from this line of work but trades syntactic expressiveness in favor of concrete efficiency to ensure practicality. Like Buffet, ZoKrates supports data-dependent control flow, e.g., allows nested loop bounds to depend on outer loops. ZoKrates supports dynamic memory for arrays but does deliberately not expose any

additional random access memory abstractions.

XJsnark [208] proposes a Java-like programming abstraction realized in the Jetbrain's metaprogramming framework for DSL development [190] that compiles to arithmetic circuits. It supports large integer data types and proposes several advanced optimization techniques, e.g., for read-only memory access and integer arithmetics, and further improves on the results. Unlike in ZoKrates, the developer is responsible for the specification of circuit-external witness derivation code (see Section 7.3.2), which is written in Java and encapsulated in special `external` code blocks. While currently not supported, long integer data types and xJsnark's algorithm for efficient read-only memory access could be implemented in future versions of ZoKrates.

TinyRAM [35] supports proving the execution correctness of programs compiled to TinyRAM assembly in zero-knowledge. The TinyRAM compiler translates a subset of C to TinyRAM assembly. Leveraging an optimized translation of arbitrary computation in the random access machine (RAM) model of computation [34], the execution of a bounded number of steps of the resulting TinyRAM assembly on the TinyRAM virtual machine is unrolled to an arithmetic circuit. By design, this circuit is satisfied by a valid execution trace. Building on this result, vnTinyRAM [38] removes the need for a program-specific trusted setup by introducing a universal circuit for the validation of tinyRAM executions fetched from memory. This modification, however, leads to significant overhead for each TinyRAM instruction. While conceptually intriguing, the resulting arithmetic circuits are extremely large, making proof generation too expensive to be considered practical even for small realistic programs.

Snarky [262] is an OCaml-based functional DSL that compiles to R1CS. The user is responsible for non-deterministic witness generation. The ZØ compiler [141] extends the C# language with special `ZeroKnowledge` blocks that contain queries to be evaluated in zero-knowledge. The compiler uses a cost model to select either Pinocchio or ZQL [138] as a zero-knowledge backend for a given query block. Adopting the idea of a distributing compiler, it produces artifacts deployable to multiple tiers for scalability.

3.3 Conclusion

In this chapter, we described and discussed work related to our core contributions and demonstrated both the novelty and relevance of these contributions in the context of the body of existing work.

Although several pattern collections have been proposed in the context of blockchains and smart contract engineering, our work represents the only pattern collection that is explicitly concerned with off-chaining and predates most other proposals. Our systematization of the design space for off-chain computations and its exploration establishes the connection between different off-chain computation approaches for the first time.

We demonstrated that ZoKrates generalizes and integrates with existing applications of zk-SNARK-based off-chain computations. Furthermore, we illustrated the need for the ZoKrates DSL by showing that existing programming abstractions for the specification of zk-SNARKs are insufficient for the developer-friendly and efficient instantiation of off-chain computations.

Part II

Off-chaining

Blockchains process transactions without relying on a trusted third party. Instead, transactions are processed by a network of equal, economically incentivized peers. Therefore, every single peer redundantly processes every transaction. After consensus on the processing results has been reached, the results become part of an agreed-on shared history. With their thereby established key properties — trustless transaction processing and immutability of the transaction record — blockchains represent a powerful and novel building block in distributed system design. However, fundamental challenges remain:

Since every peer processes every transaction, transaction processing is slow and involves significant overhead compared to other processing models. Besides low throughput and high latency, the fully redundant transaction processing implies a lack of scalability by design: Adding more nodes to the network does not increase its throughput.

Furthermore, to enable fully redundant processing, all information that is ever processed or stored by a blockchain must be shared within the network. Consequently, it also becomes part of the blockchain's public immutable history. Thus, confidential data cannot be processed by a blockchain; it would be revealed in the process. This limitation implies considerable privacy challenges faced by blockchain-based applications and their users.

In this part, we propose the concept of *off-chaining* as an approach to address theses fundamental privacy and scalability challenges encountered in the context of decentralized application design. We define off-chaining as the process of moving data and computation off the blockchain without impairing its key properties. By allowing never to publish confidential data on the blockchain in the first place, off-chaining can serve as a powerful privacy-engineering tool. Furthermore, performing computations off the blockchain and reducing the on-chain processing load reduces redundancy and thereby facilitates scalability.

In the first chapter, we identify a set of different off-chaining patterns that leverage the idea in smart contract design. From our discussion and assessment of the patterns, we conclude that the delegation of computations to blockchain-external nodes is a particularly powerful idea to address the aforementioned challenges. The chapter is based on and re-uses material from our paper published at ESOCC 2017 [112].

In the second chapter, we evaluate different possible realizations of general off-chain computations to address the conflict between privacy and on-chain verifiability through redundant processing. For that, we compare secure enclave–based, verifiable computation–based, secure multi-party computation–based, and incentive-driven off-chain computation approaches. Our analysis shows that zk-SNARK-based off-chain computations are a particularly suitable approach from a conceptual perspective. At the same time, we find that an actual instantiation of this approach comes with a set of challenges on its own and is out of reach for most developers. This finding motivates ZoKrates, our framework for verifiable off-chain computations introduced in Part III. Material used in this chapter has been previously published in our paper at SERIAL 2018 [110].

CHAPTER 4

Off-chaining Patterns

In this chapter, we introduce off-chaining as a general approach to address privacy and scalability challenges in the context of blockchains. We introduce off-chaining patterns as generic descriptions of how the concept of off-chaining can be applied. This chapter is based on and re-uses material from our paper published at ESOCC 2017 [112].

We start by providing a general definition of off-chaining. Based on our experience in building blockchain-based prototypes, we afterward identify and describe five off-chaining patterns. These patterns show how the abstract idea of off-chaining can be instantiated and leveraged to address challenges that developers commonly face during the design of blockchain-based applications. Besides the description of the patterns themselves, we supply examples and refer to existing implementations.

Before concluding the chapter, we discuss the impact of the previously introduced patterns on the privacy and scalability problems and assess the state-of-the-art with regards to their instantiation in practice. From this analysis, we conclude that the off-chain computations are a particularly powerful but scarcely developed concept. This fundamental result motivates and lays the foundation for the subsequent chapter, where we evaluate different off-chain computation approaches.

4.1 Off-chaining Definition

As described in Chapter 1, the lack of privacy and limited scalability represent core challenges faced by blockchain technologies. Any information on a blockchain is inherently public; confidentiality and privacy are and cannot be guaranteed for on-chain transactions. Furthermore, on-chain processing involves considerable overhead compared to processing on blockchain-external resources [92]. The slowest node supported by the network determines the achievable transaction validation speed , and consensus protocols add communication overhead. Consequently, there exists a cost overhead in running the shared blockchain infrastructure compared to less redundant

processing environments. Thus, miners typically charge transaction fees to compensate for their operational costs.

The limitations mentioned above directly result from the design choices made to realize blockchains' key properties. Decentralized transaction processing without requiring trust in a single party demands high redundancy, and consensus on processed transactions is required to establish an immutable history. As a result, there is no clear cut path to addressing these problems by altering the design of blockchains themselves without compromising these key properties.

Thus, instead of trying to address these inherently hard problems directly, we propose to sidestep them instead by leveraging blockchain-external resources. We refer to this idea as off-chaining, which we define as follows:

Definition (Off-chaining): *Off-chaining describes the process of moving computation or data off the blockchain — to blockchain-external resources — without losing the blockchain's desirable properties.*

By moving data and computation elsewhere off the blockchain, for example, to another datastore, server, or third party, the blockchain footprint obviously is reduced. However, the fundamental properties of blockchains and blockchain-based applications may be compromised to different degrees when doing so. They may even be prohibitively violated when using naive off-chaining approaches. Thus, in the off-chaining process, it is crucial to identify blockchain properties that are required or desirable in the given context and select an off-chaining approach that impairs these properties as little as possible. To support this selection, the remainder of this chapter discusses different off-chaining patterns that propose and implicitly categorize off-chaining approaches.

4.2 Motivational Examples

Over the last years, we gained extensive experiences in developing proof-of-concept implementations of blockchain-based applications prototypes. Together with students as well as partners from industry, we conducted a set of different blockchain development projects. We leverage this experience, combined with insights gained from studying a broad range of projects in academia and industry, to identify challenges that are recurring across different applications and approaches to addressing them.

Before we describe these solution approaches in the form of off-chaining patterns in the subsequent section, we now present two of these blockchain-based applications as motivational examples. We describe challenges in the context of these two exemplary applications and outline how they motivate and could be addressed through off-chaining.

While the prototypes for these applications have been realized on the Ethereum platform [74], we consider our findings to be representative for all of today's public, blockchain implementations that support smart contracts.

4.2.1 Fair Chess

As a first example application, we introduce a fair and manipulation-resistant chess game. An Ethereum-based prototype was developed together with a group of students and co-supervised with the Ethereum Foundation [162]. The implementation is available as open-source software [122].

Today, online gaming relies on a trusted intermediary that runs games and ensures players obey the rules. However, this intermediary needs to be trusted not to cheat or steal funds. To eliminate that trust, we implemented the chess logic as well as data structures required to persist the game state in a smart contract. Conceptually, that already solves the problem: Players send moves to the contract, which modifies the game state persisted internally for valid moves. After a valid move, the contract checks end game conditions and pays out the winner if a condition is met.

Checking end game conditions, however, is computationally expensive. All potential moves have to be calculated and verified to check the checkmate condition. This calculation is not possible in a smart contract as it violates the complexity upper bound for on-chain transactions, i.e., the gas limit in Ethereum.

Three things can be learned from that:

1. We need to find a way to perform as much of the end-game check as possible off-chain, on the client-side, without impacting the blockchain's trustlessness property.
2. As on-chain computations come with a fee, the end game check should be performed as rarely as possible. Hence, like in a physical chess game, we should have a player trigger the check instead of doing it after every valid move. In other words, we should move part of the control flow to the client-side.
3. We have to expect to reach performance limits of blockchains when creating applications. This problem emphasizes the need for research in and development of off-chaining techniques.

4.2.2 Blockchain-based Service Marketplace

As a second, more complex example, we designed a decentralized service marketplace that enables trustless disintermediation between providers and consumers of service APIs and provided a proof-of-concept implementation. Using a cryptocurrency for payments, a consumer can buy time-constrained access to a service offered on the marketplace without involving a marketplace intermediary. The results of this work have been published at ICSOC 2017 [204]

Especially service discovery, one of the main building blocks of a service marketplace, posed a significant challenge within the fully decentralized design: Data storage on blockchains is extremely costly due to full replication in the peer-to-peer network. Nonetheless, a meaningful service discovery feature requires API descriptions to be stored. Merely pointing to an off-chain reference from a smart contract, e.g., a file hosted in a cloud storage system, is no alternative to on-chain storage. This approach would introduce trust in the storage system since the data stored could change while the reference remains the same.

Additionally, since all data in a blockchain is stored on every node in the network, it is publicly visible. There is no obvious way for a service provider to hide some of her service descriptions from the public. As a direct consequence of this public visibility, there is no way to perform computations on private data on-chain without revealing it. Assume, for example, a consumer wants to prove to a provider that he has access to another provider's API. To support this, the second provider could publicly provide the hashes of all tokens that give access to his service. Then, the consumer could simply hash his private access token and show the hash to the first provider, which could, in turn, verify it by comparing it to the published hashes. However, for this to be trustless, the consumer would need to perform the hash operation on-chain and with that reveal his private token. Simply computing the hash off-chain would not prove anything to the provider.

For the same reason, encrypting data before publishing it to the blockchain also prevents it from being used in on-chain processing.[1] While this limitation may be acceptable in specific use cases, e.g., to unlock data for processing at a later point in time by then publishing the decryption key or in read-only scenarios, encryption can thus not be considered a solution to the blockchain's privacy problem.

Again, we derive three challenges from these findings:

1. We need to find a way to store data off the chain without giving up its manipulation-resistance.
2. As all on-chain data is publicly visible, techniques for trustless but privacy-preserving off-chain storage should be developed.
3. Off-chain computations on private data that can be verified on-chain without revealing said data would augment the set of possible use cases.

In summary, off-chaining strategies are needed to address both the high costs resulting from on-chain computation and storage as well as functional limitations of on-chain processing.

4.3 Off-chaining Patterns

Leveraging our previously described experience in implementing blockchain-based prototypes, we identified five off-chaining patterns that we now introduce. These patterns describe general strategies to solve common and recurring problems in blockchain-based application design. They can be used individually or in combination to move computation and data off the blockchain. Each pattern aims at maintaining the key properties of blockchains and, if required, includes techniques to ensure that they are not compromised to an unwanted degree.

For clarity of our description, we assume a smart contract abstraction to exist. Nonetheless, the patterns may also apply for more constrained on-chain execution environments, e.g., Bitcoin scripts (see Chapter 2).

[1] We deliberately do not consider the special case of fully homomorphic encryption (FHE) here, since no protocols that are sufficiently efficient to be used on-chain exist at the time of writing.

We describe each pattern following the same structure: First, we define the problem context in which the given off-chaining pattern can be used by describing an on-chain scenario. Then, we describe the solution to this problem as an off-chaining pattern, i.e., by describing which processing steps or data can be moved off the blockchain. In an example section, we provide an application example and describe how the pattern at hand can be applied. Afterwards, we discuss resulting properties and limitations. As the last part of our pattern description, we discuss implementation challenges and summarize known uses of the pattern at hand.

The off-chaining patterns introduced in this section are loosely ordered by generality. We start with patterns that address the most frequently occurring problems and then become more specific.

4.3.1 Content-addressable Off-chain Storage Pattern

Context

A large amount of data is associated with a smart contract. Storing the data on the blockchain is too expensive, but immutability is required.

Solution

Store the data off-chain in a content-addressable storage (CAS) system, i.e., a system where a record's identifier directly and verifiably depends on its content. Store that identifier in the smart contract as a reference. Clients using the smart contract can retrieve the reference from the smart contract and, based on that, retrieve the data from the CAS system. They can then verify the data's correctness by re-computing its identifier from the data itself and comparing it to the reference stored in the smart contract.

Figure 4.1 shows a schematic example of the pattern and illustrates how content-derived identifiers can be used as references in smart contracts.

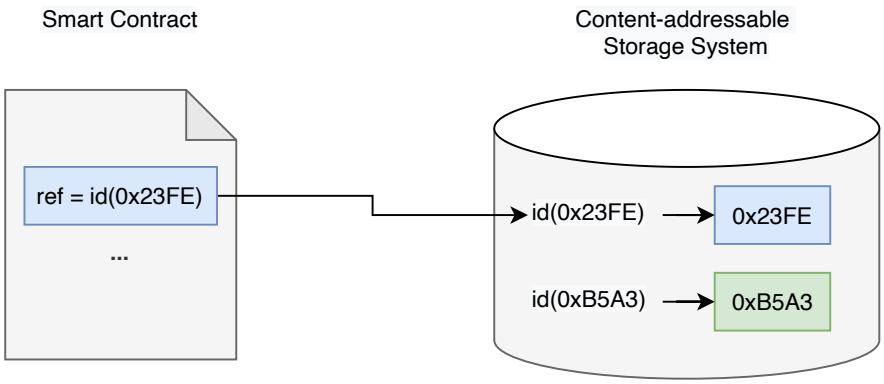


Figure 4.1: Content-Addressable Storage Pattern Example

Example

A smart contract encodes ownership of a piece of digital art. However, a piece of art would be very costly to store on-chain due to its size. Instead, the file that encodes the piece of art is stored in a CAS system that identifies files by their hashes. The file hash is stored in the smart contract, serving as a reference to the artwork. Clients can then retrieve the hash of the externally stored piece of art from the contract and use it to query the storage system. The result can then simply be hashed to verify its correctness.

Discussion

This pattern allows the trustless outsourcing of data to an off-chain storage system since a modification in the data would immediately change its address and with that invalidate its references.

By applying the pattern, an application's storage cost can be significantly reduced, and data that can not be stored on-chain in the first place can now be referenced without introducing trust. Additionally, as the data retrieval is done on the client-side from an external storage system, privacy features may be implemented by adding access control to that system. However, this requires careful considerations depending on the use case, since leaked data can immediately be confirmed to be authentic by recalculating its address.

While not in the scope of this pattern, the required external CAS system itself must be as reliable and available as possible. In case of unavailability or data-loss, the blockchain-based part of the application may also become unavailable.

Theoretically, this pattern could be extended to support trustless computation on data stored off-chain: First, content-addressed data referenced from a smart contract could be sent to the contract. Then, integrity could be verified on-chain. In case of success, the smart contract could modify the data, update its reference to that new data, and write it to an event. An untrusted external worker could then write that data back to the CAS system the inputs were retrieved from. While conceptually appealing, we did not yet observe this extension. Hence, it is not part of the pattern description. We attribute this to the high cost of on-chain integrity checks as well as the reliance on a blockchain-external worker node for liveness.

Implementation

As mentioned before, a CAS system is required to work in conjunction with smart contracts. Three such technologies, which address data by its hash and try to ensure availability and durability, are the Interplanetary File System (IPFS) [40], Swarm [324], and Filecoin [278]. Note, however, that these CAS systems do generally not provide strong availability guarantees. In IPFS, for example, a file can be lost unless there is at least one available network participant that explicitly marked it for permanent storage [279]. Swarm and Filecoin propose incentive systems to realize stronger availability guarantees; however, neither incentive system has been successfully deployed at the time of writing. Hence, data availability remains an open challenge and has to be considered in the context of implementations of this pattern.

Known Uses

The content-addressable off-chain storage pattern is widely used across many blockchain-based protocols and applications. Known uses include [65, 140, 174, 228, 271, 275, 288, 290, 304].

4.3.2 Verifiable Off-chain Computation Pattern

Context

a) A client wants to prove a property of its private data to the blockchain network without publishing it.

b) A client wants to perform a computation that is too complex to be executed within one block.

c) A client wants to reduce the cost associated with an on-chain computation.

Solution

Delegate computation to an untrusted third party and, besides the result, generate a proof of correct execution. Instead of executing the computation itself, verify the proof of correct execution on-chain.

We contrast regular on-chain processing with the processing of verifiable off-chain computations as described in this pattern in Figure 4.2: An on-chain computation is triggered by a transaction that specifies the program (for example, a smart contract function) to be run as well as its inputs. All blockchain nodes redundantly run the contained computation to process the transaction. An off-chain computation is triggered by a call to an arbitrary untrusted off-chain processing node, in which the program to run as well as its inputs are specified. This blockchain-external node then executes the computation and sends the results, as well as a proof that attests to the correctness of the computation, to the blockchain for verification within a transaction. The blockchain nodes redundantly verify the proof submitted by the off-chain node but do not re-execute the original computation.

An optimistic variant of the verifiable off-chain computation pattern even off-chains the on-chain verification step: Instead of checking the proof's correctness on-chain, the proof is only published on the blockchain, but not verified. Clients independently verify that proof and silently accept it if valid. Only on detection of an invalid proof, a client requests an on-chain check. This optimistic approach can be combined with an economic incentive design for clients that ensures that invalid proofs are reported early to avoid the need for eternal independent verification. Still, relying on such incentives weakens trust compared to direct on-chain verification.

Example

There is an on-chain list of hashes of ID-card information, which refers to people who are allowed to call a smart contract function. Anyone listed can prove that he has an ID-card, which authorizes him to call the contract function by hashing his card information locally and supplying the result,

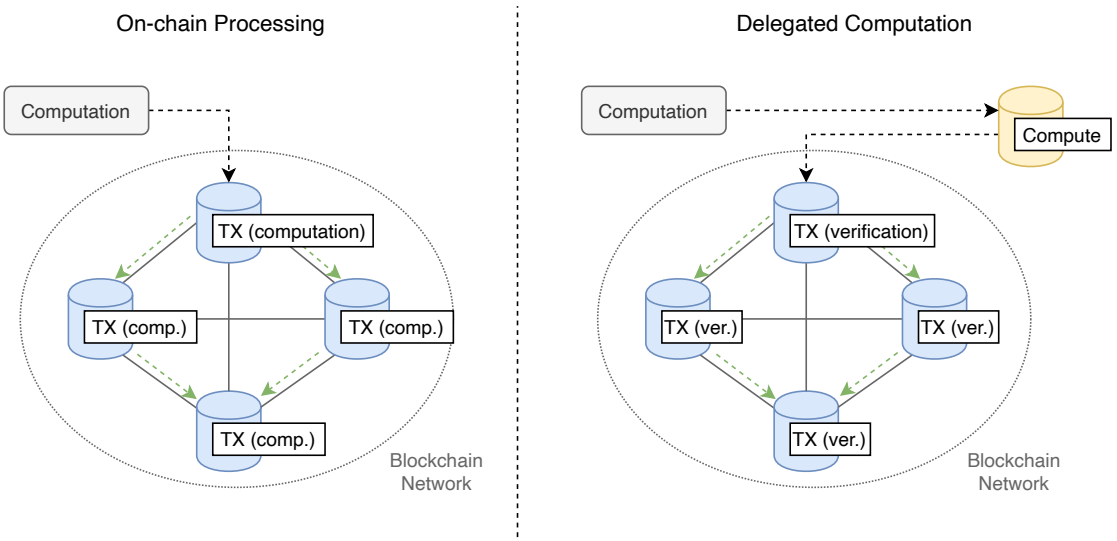


Figure 4.2: Comparison of On-chain Processing and Verifiable Off-chain Computations

including a proof that attests to the hash computation's correctness. The proof itself does not reveal any of the information on the card.

Discussion

The verifiable off-chain computation pattern allows the trustless outsourcing of on-chain computations to untrusted, blockchain-external nodes. As listed in the context section, this pattern can be instantiated to realize computations on private data to save transaction processing and to circumvent block complexity limits. Yet, these capabilities do not necessarily come together. Which ones are available for an instantiation of the pattern depends on the concrete realization.

Unlike with regular computations on the blockchain, this pattern allows off-chained computations to hide information used during execution. Hence, not having to expose information, but the result of a computation greatly enhances privacy. However, being able to use private information in a computation requires that the proof attesting to a computation's correctness does not include said information or reveal anything about it. As introduced in Chapter 2, protocols that support proofs of statements that involve secret inputs — private information — are called zero-knowledge protocols. Privacy-preserving off-chain computations can be built from zero-knowledge protocols and ensure that no private information is published to or recoverable from the blockchain's transaction history. The only information leaked by an off-chain processing node is the fact that it has all the information necessary to compute the output correctly. It does not have to reveal any private inputs or intermediate results of the off-chain computation.

If a client decides to act as the off-chain computation node itself, the private information never leaves infrastructure controlled by the client. No trust in external parties is required. However, if a client decides to delegate an off-chain computation to a third party, it does have to trust that third party not to reveal its private information. Correctness guarantees are not affected.

Besides incorporating private data in blockchain-based computations, saving transaction fees can be a reason for the adoption of the off-chain computation pattern. If an off-chain computation scheme can be designed in a way that the verification of a computation's correctness on the blockchain is cheaper than the on-chain execution of the computation, fee savings apply.

Furthermore, if the off-chain computation pattern can be instantiated in a way that the verification complexity is independent of the complexity of the off-chained computation, another interesting property results: Even computations exceeding the on-chain complexity limits (e.g., the gas limit in Ethereum) may still be executed off-chain. Thus, this approach can be leveraged to increase blockchains' throughput. Additionally, the constant verification fee implies an upper bound for any computation's processing fee.

Implementation

Implementing off-chain computations is not trivial. To be able to verify off-chain computations from smart contracts, the underlying blockchain needs to support the operations needed to check proofs. These can either be use case–specific or universal building blocks, which can be used to verify any proof. The Ethereum blockchain, for example, added cryptographic primitives to support the verification of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) over a specific elliptic curve [27] to the Ethereum Virtual Machine (EVM) with the Ethereum Improvement Proposals 196 and 197 [130]. This extension is lightweight and can easily be added to other blockchain implementations as well.

As previously discussed, the benefits resulting from this pattern's instantiations depend on its concrete implementation. In Chapter 5, we will generalize these results to arbitrary use cases and analyze generic implementation options and how they compare in detail.

Before the introduction of ZoKrates, which we present in Part III of this thesis, implementing verifiable off-chain computations required significant cryptographic expertise and was simply out of reach for most decentralized application (dApp) developers.

Known Uses

Zcash [177] uses a zk-SNARK-based verifiable computation scheme together with a custom blockchain derived from the Bitcoin implementation to implement an anonymous currency. Inspired by the Zerocoin [245] and Zerocash [293] protocols, Zcash uses zk-SNARKs to implement off-chain computations that incorporate private spending keys. This implementation is specifically designed for anonymous token transfers and cannot easily be generalized for arbitrary off-chain computations.

Additionally, known uses of ZoKrates (see Part III) represent instantiations of this pattern, e.g., [30, 143, 169, 175, 267, 277, 307, 314].

4.3.3 Off-chain Signatures Pattern

Context

Two network participants know that they will perform a set of transactions in the future. They want to reduce the cost of these transactions or want to hide them from other network participants.

Solution

Together, the two participants specify and agree on a smart contract, including a function, which applies an external state given as an argument to the contract state. This function includes a signature check to ensure both participants agree with the state change: Only if valid signatures of both participants are supplied with a requested new state, the new state is applied. This contract is deployed to the blockchain, and both participants optionally make a deposit.

Then, the participants transact purely off-chain and peer-to-peer, without involving the blockchain: One participant computes a new state, wraps it in a message, signs it, and sends it to his counterpart. The recipient then checks the new state, signs the message as well if he agrees, and sends it back to the sender.

This message, signed by both parties, can now be sent to the smart contract by a participant at any point in time. After validating both signatures, the contract updates its state accordingly. The smart contract allows an updated state to be disputed by sending a newer state for a period of time before it is finalized. This dispute period discourages malicious participants from publishing outdated states that benefit them. In case a recipient refuses to sign a valid new state, the original sender can publish the last known state to the blockchain and then trigger the state transition in an on-chain transaction.

Example

Participants A and B create a smart contract with a signature-locked state update function and deposit 50 units of cryptocurrency each. Now, A wants to transfer 10 units to B. For that, she creates a transaction locally, which includes a new state where A and B have balances of 40 and 60. She signs it and sends it to B, who signs it as well. Now, B can use the transaction to update the on-chain balance at any point in time. However, A and B could perform further off-chain value transfers without ever settling on-chain unless one side's deposit is used up. The previously described instantiation of the pattern for off-chain value transfers is often referred to as a payment channel. The generalization to arbitrary states has been referred to as state channel [88].

Discussion

The off-chain signatures pattern enables efficient off-chain transactions between a set of participants without re-introducing trust into the system. The core insight is that the guarantee to be able to settle an agreed-on off-chain transaction at any time is as good as actually executing the transaction on-chain. Signing a new state is analogous to writing a check in a traditional financial transaction. Using off-chain transactions can lead to significant cost savings as transaction fees

only apply for on-chain settlement. Furthermore, the pattern can enhance privacy and confidentiality since all transactions except for the final settlement remain hidden from the network. From a blockchain network's perspective, this pattern reduces the load on the system and thus enhances scalability by freeing up capacity for other tasks.

There are many other applications besides simple value transfers. As shown in Figure 4.3, we were able to move the core parts of the chess game off-chain by using this pattern, for example (in combination with the optimistic finalization pattern described in Section 4.3.4 for end-of-game checks). In this context, the off-chains signature pattern not only helped to lower the cost of a game but also ensured the game feels real-time and no longer depends on block intervals. The blockchain's role is reduced to settling disputes — in case any arise — and to optionally pay a price to the winner. For more information on this specific example, refer to [122, 162]. In a similar spirit, a recent case study by McCorry et al. presents a state channel–based battleship game implementation [237].

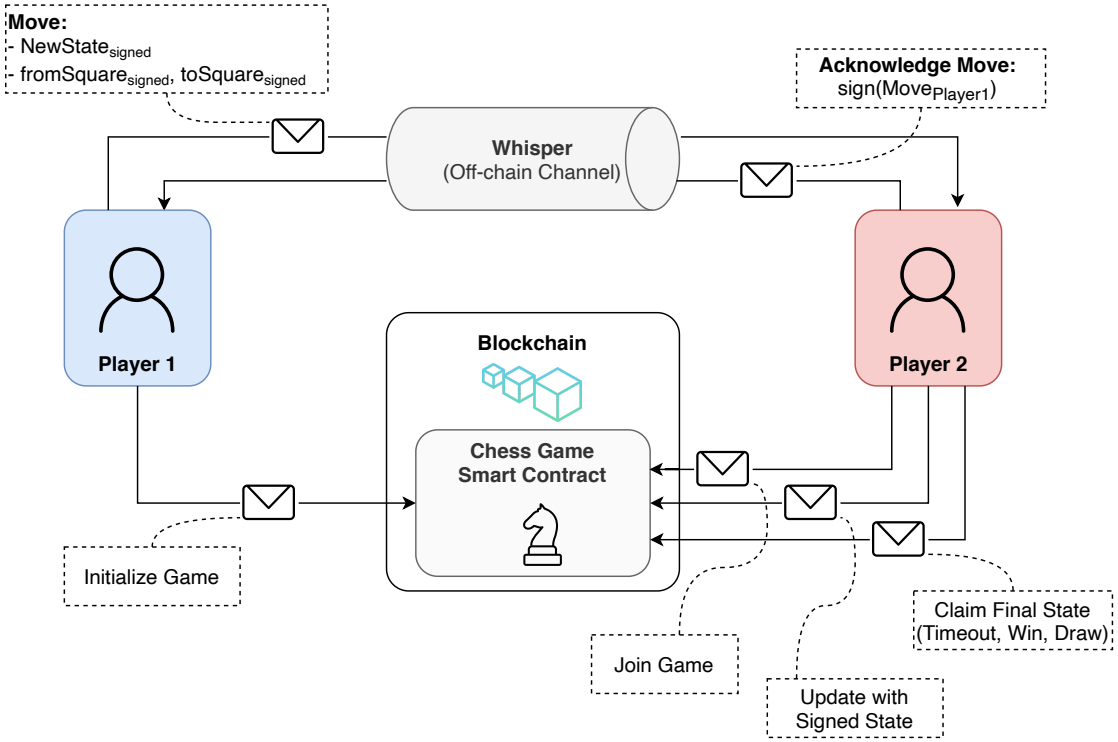


Figure 4.3: Off- and On-chain Interactions in the Chess Application Example

Since initial deposits to the smart contracts are required in most cases, establishing contracts with many peers can lock a considerable amount of funds. Also, malicious participants could freeze funds by denying signatures. Hence, contracts should specify timeouts that trigger automatic settlement. These timeouts should be sufficiently large to allow participants to respond even in case of blockchain network congestions. Thus, timeout-based approaches require participants to be highly available. Furthermore, in case of the temporary unavailability of a participant, a mali-

cious counterparty could settle an old state beneficial to the attacker on the blockchain. Crypto-economic delegation protocols have been proposed to address this issue [236]. An asynchronous construction that does not rely on timeouts was proposed by Avarikioti et al. [20].

Implementation

Besides on-chain smart contracts, this pattern requires a peer-to-peer communication channel to exchange signed messages off-chain. This communication channel should be chosen depending on application requirements. In cases where anonymity is not a major concern, direct peer-to-peer channels can be used. For additional identity protection, mixnets [81] or other protocols for anonymous message exchange, e.g., the broadcast-based Whisper protocol [131] in the Ethereum ecosystem, should be considered. Unger et al. provide an overview of state-of-the-art secure messaging options [328]. For off-chain signature-based payments, Green and Miers proposed an anonymous-by-design payment channel constructions [163].

Known Uses

The most prominent known uses of the off-chain signature pattern occur in payment channel networks, which build off-chain value transfer networks using existing blockchains for settlement: The Lightning Network [276] provides an implementation for the Bitcoin ecosystem, while Raiden [60] targets the Ethereum network.

State channel-based game implementations, e.g., battleship [237] and chess [162], represent other known uses of the off-chain signatures pattern.

4.3.4 Optimistic Finalization Pattern

Context

A smart contract implements a state machine that has a set of well-defined final states. State transitions are cheap to compute, but checking whether a given state is a final state is expensive.

Solution

Instead of checking whether a state is final or not in a smart contract on a blockchain, the same check is performed off-chain on the client-side. A client can assert that a final state has been reached to the smart contract. The blockchain does not verify this claim but stores it. Other clients can invalidate the finality claim by responding to the blockchain with a valid state transition within a pre-defined response period. Using this pattern, the computation that checks state finality never has to be performed on-chain. A response by a client can also signal agreement with the initial finality claim. Such responses allow the decision regarding the claim to be finalized before the response period times out.

We provide an overview of this pattern and its steps in a flowchart in Figure 4.4.

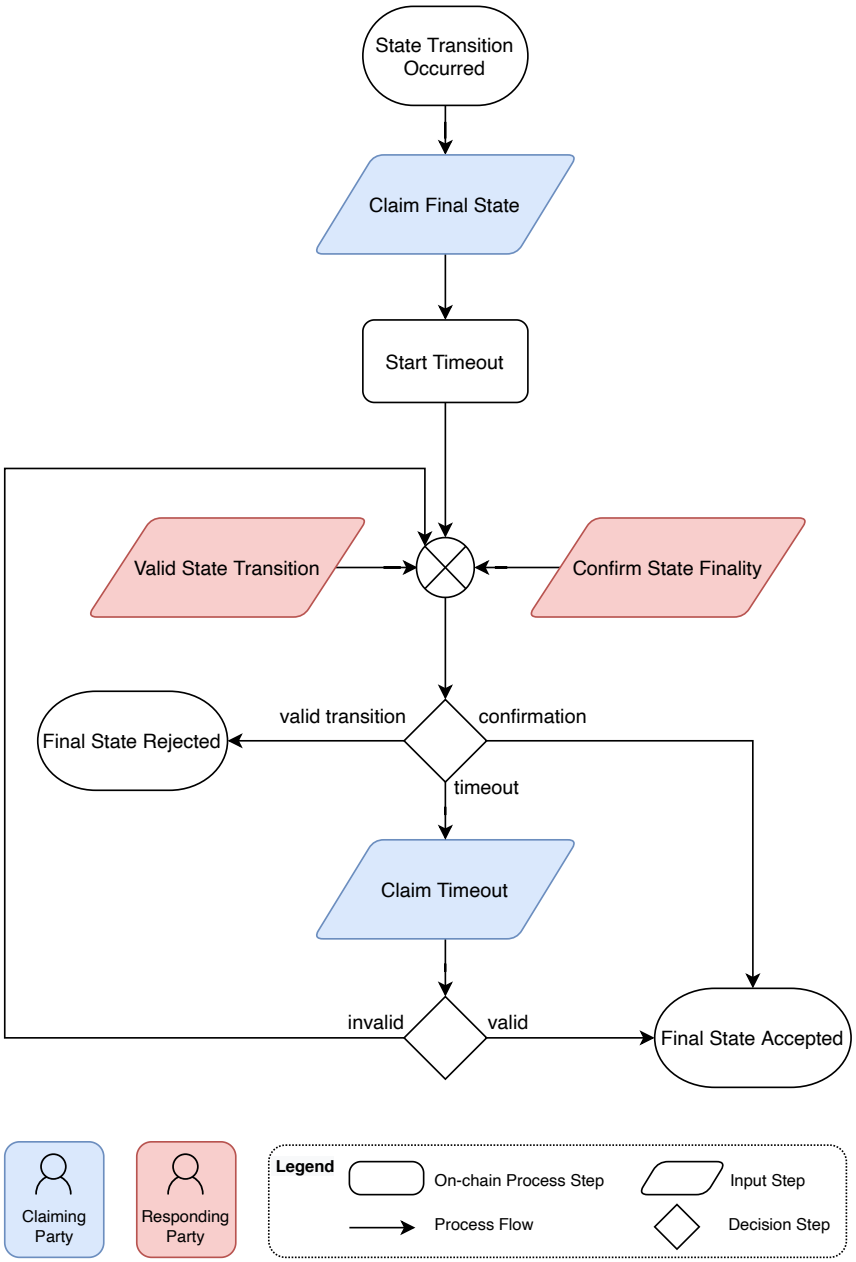


Figure 4.4: Flowchart representing the Optimistic Finalization Process

Example

The end game condition for chess is too expensive to check on-chain. The players, however, can easily check the condition off-chain. Hence, instead of triggering the end game condition check in a smart contract, a player simply claims checkmate. If the claim is false, his opponent can simply prove him wrong by submitting a valid move. If the claim is true and the opponent cannot submit a valid move, the winner is paid out after a dispute period has passed[2].

In Figure 4.5, we illustrate how we use the optimistic finalization pattern in the context of our chess application example described in Section 4.2 also considering draws, stalemates and timeouts. For an extensive treatment of the design and implementation, refer to [122, 162].

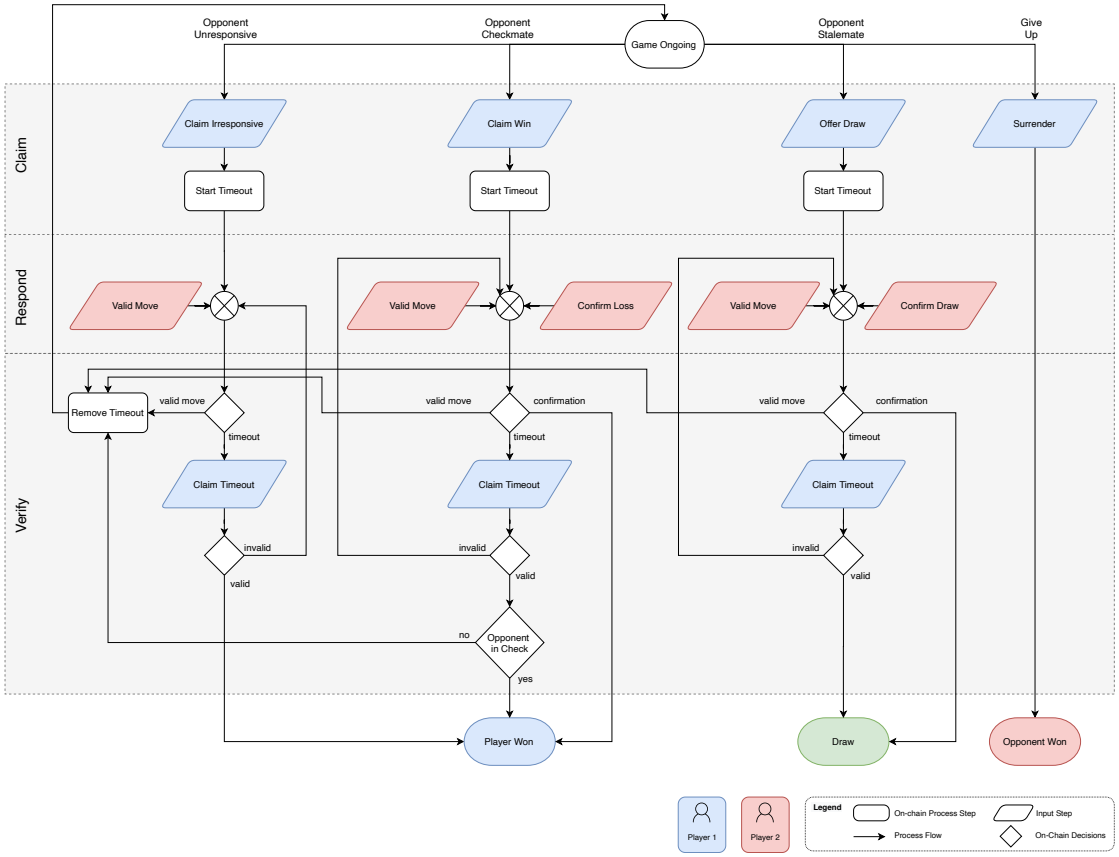


Figure 4.5: Optimistic Finalization Pattern applied to Chess Example

[2] This is slightly simplified: To ensure the final state is actually checkmate and not stalemate, the blockchain checks whether the opponent is in check or not (see Figure 4.5). This step can partially be off-chained by asking the claimant to point out the piece that gives check and only verifying that claim on-chain.

Discussion

This pattern allows computations that check state finality to be off-chained efficiently in scenarios where smart contracts act as state machines. Since it allows for complex operations to be moved entirely off-chain and with that circumvents the complexity upper-bound for on-chain transactions, it can extend possible use cases and potentially lead to cost savings. Note, though, that the pattern increases the overall amount of on-chain transactions and thus requires a careful assessment of the overall cost impact. Also, increased availability of the parties involved in the smart contract implementing the pattern is required, since the use of timeouts is essential to ensure progress.

In the previously described pattern, the blockchain could judge whether a state was final based on the absence of a valid response. More precisely, a claim was regarded as valid if there was no invalidating response within the dispute period. By designing claims so that observers can produce invalidity-proofs that are cheap to verify on-chain if required, we can generalize the optimistic finalization pattern to realize off-chain checks of arbitrary statements, even complex off-chain computations. While this idea is intriguingly powerful, designing claims so that cheap-to-verify invalidating responses exists is not trivial. Instantiations of this idea, e.g., Truebit [320] or Arbitrum [193], come with a whole set of challenges and limitations on their own as we discuss in more detail in Chapter 5. Although this generic concept is far beyond the scope of a concrete, instantiable pattern, we do encourage readers to keep it in mind as an inspiration and an abstract approach for designing creative off-chaining solutions.

Implementation

This pattern does not require additional technologies besides smart contracts. For an exemplary implementation of this pattern, refer to [122].

Known Uses

Like in the case of our chess example [162], the optimistic finalization pattern is used to decide on states in the design of blockchain-based protocols in [98, 104, 235, 338].

4.3.5 Low Contract Footprint Pattern

Context

Changing a smart contract's state requires an on-chain transaction. To incentivize the blockchain network to process transactions, a fee has to be paid. This fee depends on the complexity of the called smart contract function as well as its use of storage. Users strive to minimize costs.

Solution

Contracts should be designed in a way that minimizes the number, complexity, and size of on-chain transactions to optimize fees. The following two techniques can be used to reduce the footprint.

- Do not check conditions on-chain after a state change. Let nodes perform the condition check locally and trigger an on-chain check in case of success.
- Optimize for writes, not reads. Reading from a smart contract is a local off-chain operation and does not require an on-chain transaction. Minimize writes and store information free of redundancy. Compute derived data locally during reads.

Examples

- In the service marketplace application, a service provider needs to make sure consumers are removed from the on-chain authorization list after the period of time the consumer paid for is over. Instead of periodically triggering or linking the condition check to another contract function and risking frequent re-evaluation, he tracks the access period locally and triggers the on-chain check after it has elapsed. This strategy reduces the amount of on-chain evaluations to one.
- If the service provider wants to know the number of customers currently subscribed to his service, he should not add a counter to the smart contract. He can compute the number locally at any point from the authorization list. This strategy saves storage space and counter update operations.

Discussion

This pattern may not initially seem like an off-chaining approach, as it does not explicitly move computation or data off the chain. However, by altering smart contract interfaces as well as the way they are invoked, it prevents information from being stored or processed on-chain in the first place and thereby reduces associated costs. To further reduce fees, the low contract footprint pattern can be combined with on-chain efficiency optimization strategies as presented by Brandstätter et al. [61], for example.

Implementation

No additional components or techniques are required besides smart contracts to implement this pattern. The ideas presented in this pattern should be incorporated in any thoughtful smart contract design.

Known Uses

OpenZeppelin instantiates the low contract footprint pattern in a payment contract [264] of their widely used library for secure smart contract development. In a similar spirit, ConsenSys proposes an instantiation of this pattern as a smart contract engineering best practice [101].

The Ethereum Alarm Clock [242] off-chains control flow by allowing Ethereum transactions to be scheduled for later, time-triggered execution.

4.4 Impact and Open Questions

We now briefly discuss how the previously described patterns impact scalability and privacy in the context of blockchains. In the process, we identify open questions that motivate further research.

4.4.1 Scalability

The content-addressable off-chain storage enables the outsourcing of data from the blockchain to external storage services without re-introducing trust with regards to data integrity. Thus, the pattern provides a means to keep the blockchain's state size as small as possible and reduces the load for nodes in the blockchain network. The amount of data that immutability can be guaranteed for by a blockchain network increases by orders of magnitude. An open question is how the availability of off-chain data, which is stored with less redundancy, can be ensured. Established CAS systems do not offer strong availability guarantees (see Section 4.3.1). CAS systems that ensure availability through economic incentives, e.g., Filecoin [278] and Swarm [324], are promising proposals but have not been successfully deployed at the time of writing.

The verifiable off-chain computation pattern allows a blockchain to act as a verifier of an off-chain computation instead of executing the computation itself. If verification is cheaper than on-chain execution, the overall computational complexity that can be processed by a blockchain can be increased by leveraging the pattern. Furthermore, the block complexity limit can be avoided through instantiations, where the on-chain verification cost is independent of the off-chain computation's complexity. However, such instantiations are non-trivial and come with their own set of limitations, as we analyze in Chapter 5. How the pattern can be instantiated and which instantiations are suitable for a given use case are open questions we try to answer therein.

The off-chain signature pattern allows a set of participants to execute protocols fully off-chain after an initial on-chain registration step. They only involve the blockchain in case liveness is impaired (for example, the opponent in a chess game does not acknowledge a move) or they require on-chain actions to be taken (for example, payouts to the winner of a chess game or closing of a payment channel). In case all participants behave, the blockchain does not intervene. This off-chaining approach is optimistic since malicious protocol participants can force the processing to happen fully on-chain. It is an open challenge to find game-theoretic designs that disincentivize malicious actors from forcing on-chain processing.

By shifting (potentially prohibitively) expensive conditional checks to clients, the optimistic finalization pattern can reduce blockchain processing load and even enable checks that were not possible otherwise. However, responses to claims and the use of timeouts require additional transactions. Hence, the scalability impact of this pattern has to be assessed per use case.

Unlike the other patterns, the low contract footprint pattern does not employ blockchain-external resources but helps to avoid unnecessary on-chain operations by changing control flow. Thus, the scalability benefits obtained from applying this pattern in isolation are relatively small. Nevertheless, this pattern remains important to minimize transaction fees for users.

In summary, blockchain scalability can significantly benefit from off-chaining, as described by our patterns. By combining these patterns, computation and data can be moved off the blockchain, which frees up on-chain capacity. Importantly, desirable blockchain properties are not significantly impaired in this process.

4.4.2 Privacy

The content-addressable off-chain storage pattern allows replacing on-chain data with references to data stored on blockchain-external resources. Whoever obtains that data can check its integrity by locally re-computing its content-based address and comparing it to the on-chain reference. This check does not require revealing any data to the blockchain. Hence, the content-addressable off-chain storage pattern allows referencing private off-chain data while ensuring immutability. This data can not be used in on-chain processing, as this would require it to be revealed to the blockchain first for an integrity check. Access to potentially private off-chain data is controlled by the CAS system's access management.

Instantiations of the verifiable off-chain computation pattern can offer unique privacy capabilities as they allow proving the correctness of computations on data that remains private to the prover. The verifiable computation scheme chosen for the instantiation needs to have a zero-knowledge property for this strong privacy guarantee to apply. Instantiations, where correctness proofs are not zero-knowledge, are still viable for scalability purposes but have to be ruled out as privacy-engineering tools. Despite its significant potential, implementations of this pattern are still rather rare and use case–specific, e.g., Zcash [177]. In the subsequent chapters of this thesis, we analyze the reasons for the slow adoption and address those by enabling and simplifying efficient implementations of this pattern for a wide range of use cases.

The off-chain signatures pattern allows a protocol to be executed fully off-chain after an initial on-chain registration step. If all participants behave honestly and no errors occur, only the final state of the protocol is published to the blockchain on completion. However, a malicious participant can leak any intermediate protocol states that result from off-chain state transitions. Such intermediate states may explicitly or implicitly reveal information that otherwise would have never become public. Consider a payment channel between Alice and Bob, for example: Both make an initial deposit of 50 units in the on-chain registration step resulting in the balances Alice: 50 and Bob: 50. They then transact off-chain and, at some later point, close the channel through an on-chain transaction. This transaction applies the latest distribution of funds, for example, Alice: 30, Bob: 70. A blockchain-observer does not learn anything about the individual transactions between Alice and Bob that occurred between opening and closing the channel. Only the fact that a net amount of 20 units was transferred from Alice to Bob is revealed. Still, Alice and Bob can leak the set of intermediary states to external parties. The contained signatures allow these external parties to confirm authenticity. From the intermediary states, it is trivial to recover the exact list of transfers between Alice and Bob. Due to this risk, a careful assessment of privacy requirements and trust assumptions is needed when instantiating the off-chain signatures pattern.

The optimistic finalization pattern allows off-chaining of a conditional check that uses on-chain state as its only input. Since, by definition, check incorporates only information on the blockchain,

it cannot be used in a privacy-enhancing way.

The low contract footprint pattern defines techniques to avoid unnecessary on-chain transactions, including computational steps or storage, to save processing fees. While the pattern may help to avoid superfluous information from being published, it does not offer technical means for privacy protection.

In summary, privacy concerns can be addressed by three off-chaining patterns to different degrees:

(i) The content-addressable off-chain storage pattern allows referencing private data stored in an external system. Immutability and integrity are ensured and can be checked by anyone in possession of the data. Hence, it should be considered a standard approach when addressing privacy concerns. However, the data cannot influence on-chain state.

(ii) The off-chain signatures pattern does allow data that is only shared within a group of protocol participants to affect on-chain state. It does not need to be published on the blockchain. For this data to remain private, participants need to trust that none of them ever reveals it. Thus, this pattern's privacy guarantee is based on a strong trust assumption. Still, it can be considered an improvement compared to on-chain processing.

(iii) The verifiable off-chain computation pattern allows inputs that remain a secret to an off-chain node to influence on-chain state. It enables computations on private data without losing public verifiability as a key property that establishes trust in blockchains. For this property to materialize, an instantiation has to select a suitable verifiable computation scheme that supports zero-knowledge proofs. Since private information does never have to leave the off-chain processing node, this pattern provides a particularly strong privacy guarantee based only on cryptographic assumptions (see Chapter 5 for an analysis of such assumptions).

4.5 Conclusion

In this chapter, we introduced off-chaining as an approach to move data or computation from the blockchain to blockchain external resources to address scalability and privacy challenges. We introduced five off-chaining patterns that allow off-chaining in specific problem contexts without compromising the blockchain's desirable properties. Seen from a more holistic point of view, our off-chaining patterns provide means to reduce the blockchain's footprint in decentralized applications.

We showed that off-chaining static data to blockchain external storage is feasible and benefits both scalability and privacy. Since state-of-the-art CAS systems do not provide strong availability guarantees, however, availability of off-chain data remains an open issue that has to be carefully considered.

Realizing off-chaining computations without re-introducing trust, however, is a significantly harder problem. Still, off-chain computations are highly desirable as they do not only benefit scalability but can provide strong privacy guarantees (as demonstrated in the verifiable off-chain

computation pattern, for example). Suitable instantiations enable data-dependent smart contract operations without exposing the data the operation depends on.

We conclude that a combination of both off-chain storage and privacy-preserving off-chain computations would enable engineers to build scalable decentralized applications with strong privacy properties. To gain more insights on possible instantiations of off-chain computations beyond our off-chaining patterns, including their properties, tradeoffs, as well as concrete implementation options, we analyze approaches to off-chain computations in detail in the next chapter.

CHAPTER 5

Approaches to Off-chain Computations

In the previous chapter, we introduced the concept of off-chaining and described patterns that can be used to outsource data and computations to blockchain-external resources in specific problem contexts. We showed that generic and practical strategies for data off-chaining exist and discussed implementations. Our analysis revealed that off-chain data storage alone is insufficient to fulfill applications' privacy and scalability requirements. Only in combination with off-chain computations, scalable processing with strong privacy guarantees can be realized. While our patterns show how off-chain computations could be implemented in specific contexts (for an example, see Section 4.3.4), they do not include solutions for the more complex general case. Suitable instantiations of generic off-chain computations remain an open question.

To address this question and to gain further insights on the applicability and potential of generic off-chain computations, we provide an in-depth description and analysis in this chapter: We identify and evaluate four different fundamental approaches towards off-chain computations, namely verifiable, enclave-based, incentive-driven, and secure multi-party computation–based off-chain computations. We compare implementation options and contrast them with regards to their security assumptions, programmability, scalability properties, and privacy guarantees. From our analysis, we conclude that zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK)-based verifiable off-chain computations are a particularly powerful approach to address scalability and privacy concerns in decentralized applications. However, we find that this approach is hard to instantiate, and programmability is a major concern. Material used in this chapter has been previously published in our papers at SERIAL 2018 [110] and IEEE Blockchain 2018 [113].

5.1 Off-Chain Computations

Not every computation that is processed outside of an on-chain transaction is considered to be an off-chain computation. This naive definition would introduce trust in the party that executes the computation on blockchain-external resources. What if the node executing the off-chain computation lies about the result? Such off-chain computations would barely be useful. Public verifiability as a key blockchain property would be lost, and our off-chaining definition given in Section 4.1 violated.

5.1.1 Definition

For off-chain computations to be meaningful, it is of utmost importance to ensure that only correct computation results are ever accepted. This correctness property is ensured through off-chain computation protocols, which we define as follows:

Definition (Off-chain Computation Protocol): *An off-chain computation protocol is a protocol that allows the execution of computations on blockchain-external resources and guarantees (under protocol assumptions) that the correct processing results are available on the blockchain on protocol completion.*

Note that, according to our definition, correctness does not have to be directly verifiable by the blockchain itself. It is sufficient for a protocol to convince an independent blockchain observer. Thus, correctness can be an emergent property of an off-chain computation protocol that allows invalid intermediate on-chain results.

Building on our previous definition of off-chain computation protocols, we can capture the notion of off-chain computations more precisely:

Definition (Off-chain Computation): *An off-chain computation is a computation executed in the context of an off-chain computation protocol that guarantees correctness of accepted on-chain results. It is specified through an off-chain program and a set of inputs.*

5.1.2 Basic Architecture

We provide a conceptual overview of a basic off-chain computation architecture, which details the components that are usually involved in off-chain computation protocols and their relationship, in Figure 5.1. By establishing a terminology independent of specific off-chain computation approaches, this overview lays the foundation for their introduction in subsequent sections.

Two components are always part of an off-chain computation architecture, an off-chain node, and a blockchain network. From a technical perspective, the off-chain node can be an arbitrary machine that fulfills the requirements defined by a given off-chain computation protocol. This node executes programs for given inputs — the off-chain computations — and produces a set of *outputs* as a result. We distinguish two types of inputs: We call inputs that remain a secret to the off-chain node *private inputs*. They are not published to the blockchain or any other party with

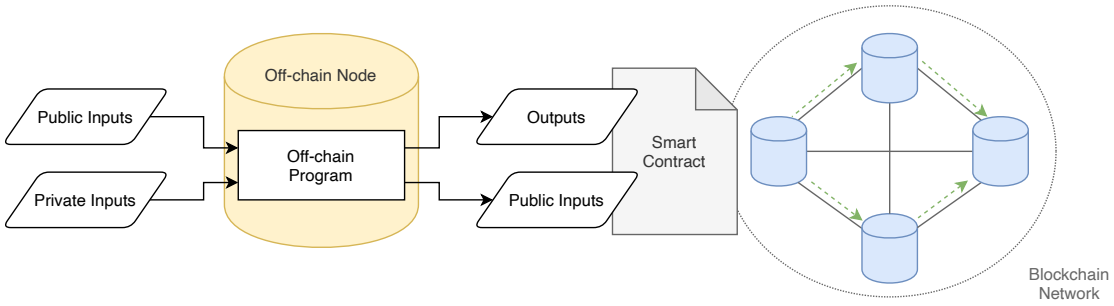


Figure 5.1: Basic Off-chain Computation Architecture

the off-chain computation's result. In contrast, *public inputs* are revealed to the blockchain like outputs.

The blockchain hosts a smart contract that receives off-chain computations' outputs and public inputs.[1] Its specific design and capabilities depend on the concrete off-chain computation protocol employed. On completion of this protocol, it is guaranteed that the smart contract stores the correct result, i.e., public inputs and outputs of a given off-chain computation.

5.1.3 Design Space

Now, after having established basic definitions and terminology, we briefly explore the dimensions of the design space for off-chain computation protocols. We introduce three dimensions that characterize and determine properties of such protocols and help to understand tradeoffs in off-chain protocol design.

An off-chain computation protocol can be categorized by making a choice in each of the following three dimensions:

Sanguinity

On receipt of a computation result by the blockchain, the blockchain can try to verify the result or optimistically accept it. In the second case, the blockchain would only attempt verification if explicitly requested by a third party. We call protocols that always verify *pessimistic*. In contrast, *optimistic* protocols, representing the opposite manifestation of this dimension, only verify on request.

Interactivity

We call off-chain computation protocols where an off-chain computation node can convince the blockchain of its computation result's correctness in one message *non-interactive*. In contrast,

[1] Conceptually, off-chain computation approaches are also compatible with blockchains that do not support smart contracts. For clarity of presentation, we limit our analysis to smart contract–enabled blockchains without loss of generality.

protocols that involve multiple rounds of messages between the blockchain and a set of off-chain nodes are referred to as *interactive*.

Responsibility

The blockchain can have different roles and responsibilities with regards to verification in off-chain computation protocols: It can act as a verifier that directly checks and decides on the correctness of an off-chain computation. If the verification succeeds, the result is accepted, and the off-chain computation node was right. We refer to such protocols as *accepting*. Direct verification is not necessarily required, however. Alternatively, a result can be accepted when it could not be proved wrong. In interactive protocols, for example, where multiple conflicting result claims are sent to the blockchain, it is sufficient for the blockchain to act as a *judge* that reliably identifies one of two conflicting claims as incorrect for the protocol to make progress. Here, correctness of the off-chain computation protocol is an emergent property that is never directly verified but indirectly results from rejecting invalid claims. Thus, we refer to such protocols as *rejecting*.

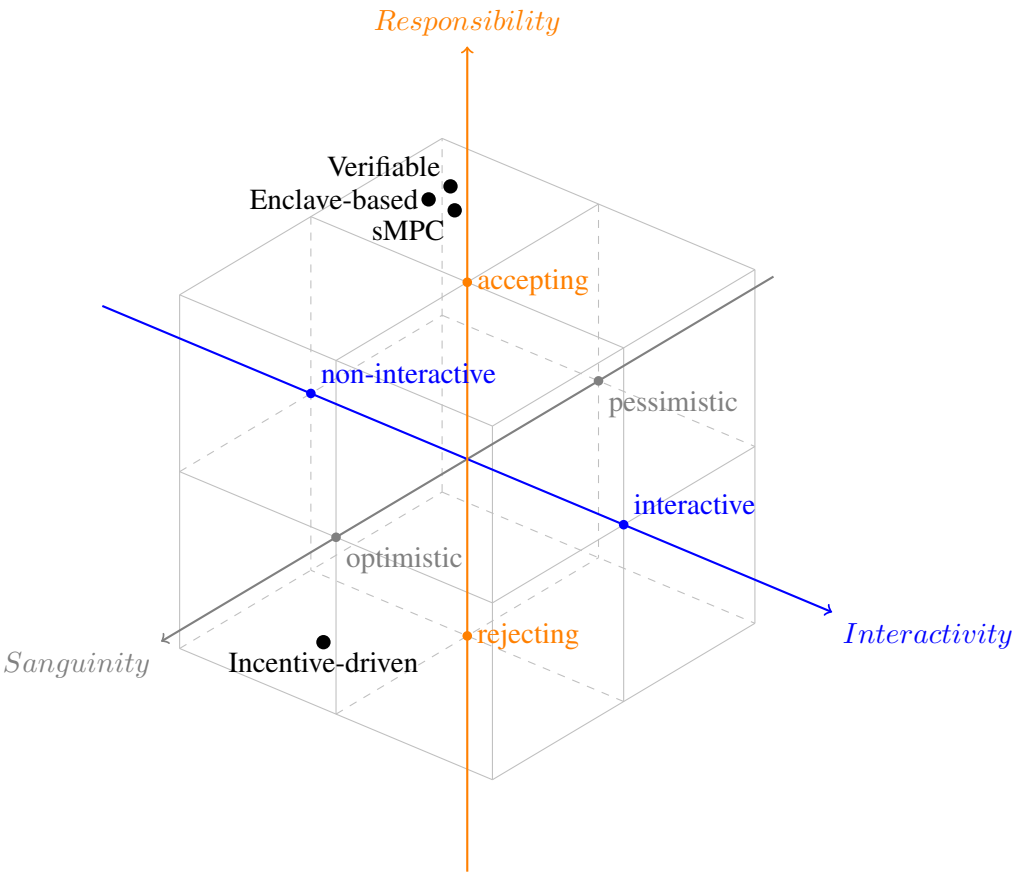


Figure 5.2: Off-chain Computation Protocol Dimensions and Manifestations

5.2 Off-Chain Computation Approaches

In this section, we present approaches that allow off-chain computations without introducing strong trust assumptions.

An approach groups a set of conceptually similar off-chain computation protocols that share a common architecture. We describe the conceptual idea for each approach before demonstrating how the basic off-chain computation architecture depicted in Figure 5.1 is adapted to support specific requirements. Then, we describe concrete realizations in the form of protocol proposals and implementations.

5.2.1 Verifiable Off-chain Computations

As the first off-chain computation approach, we present verifiable off-chain computations. Verifiable off-chain computation protocols are accepting protocols that are usually pessimistic and non-interactive.

Concept and Architecture

The core conceptual idea of the verifiable off-chain computation approach is that an off-chain node not only executes a computation but also generates a proof that attests that computation's correctness. The blockchain can then verify the computation's result through the associated proof. Proofs of computational correctness, as required by this approach, can be realized through verifiable computation schemes as described in Section 2.2.1.

We depict the basic architecture that is shared by realizations of the verifiable off-chain computation approach in Figure 5.3. The off-chain computation node acts as a *prover* that executes a computation by running a program on a set of public and private inputs, generates a cryptographic proof attesting the computation's correctness, and then publishes the result including the proof and public inputs to the blockchain. Here, the verification logic is specified in a verification smart contract. This contract acts as a *verifier* that checks the proof and accepts the result in case of success.

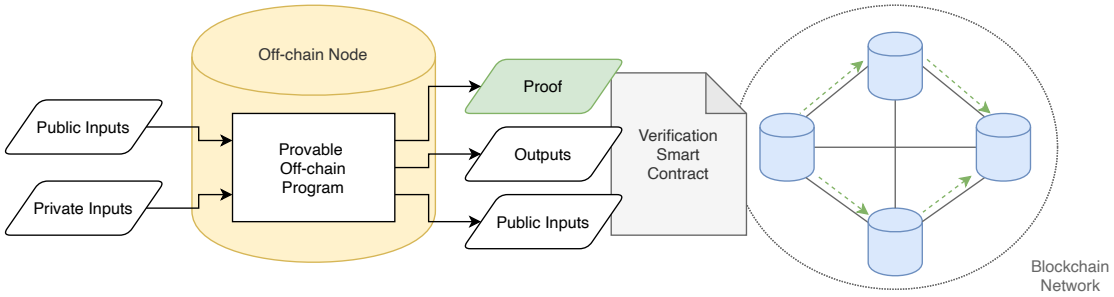


Figure 5.3: Architecture of Systems that realize the Verifiable Off-chain Computation Approach

As the verifiable computation scheme represents a central building block in this approach, its choice largely determines the core properties of a realization. Thus, we identify a set of require-

ments that we consider to be especially relevant in that context before discussing realization options and their specific properties. Refer to Section 2.2.1 for definitions. Depending on the use case, not all of the following requirements do have to be fulfilled by a suitable scheme.

Zero-knowledge: To support private inputs, i.e., inputs to an off-chain program that remain secret to the prover, the proofs generated in the context of a verifiable computation scheme need to have the zero-knowledge property. Zero-knowledge proofs reveal nothing but the fact that is proved (see Section 2.2.1 for more details). While desirable for privacy and confidentiality, this property is not crucial to realize scalability benefits through off-chain computations.

Non-interactivity: For efficiency, a prover should be able to convince a verifier in one message by providing cryptographic proof. Interactive schemes requiring multiple messages imply multiple blockchain transactions, which increases the load on the blockchain network and increases verification cost. Many interactive protocols can be made non-interactive by applying the Fiat-Shamir Heuristic [133].

Efficiency: On-chain verification should be cheap compared to native on-chain execution. Otherwise, there would be no scalability benefit. If confidentiality, however, was the motivation for off-chaining a computation, additional cost over on-chain execution can be acceptable. The two factors influencing the cost of on-chain verification are proof size and verification complexity. Thus, proofs should be as short and verification as simple as possible. The cost of verification replaces the cost of on-chain execution of the computation in the first place. High verification complexity causes higher fees, and there exists an upper bound for a block's complexity. Ideally, verification cost is independent of the computational complexity of the off-chain computation. Similarly, large proofs that need to be submitted to the blockchain cause additional cost. Since blockchains limit the amount of data that can be sent in a block, there exists an upper bound for proof size. Proof generation is usually the expensive step in verifiable computation schemes. Here, the asymmetry in on-chain and off-chain execution environments benefits efficiency: The cheap step is performed redundantly on-chain, while the expensive step can happen in an off-chain environment on powerful hardware without redundancy. Still, proof generation has to be reasonably cheap to ensure practicality of the overall approach.

Weak Security Assumptions: Security assumptions should be as weak as possible. Strong assumptions require a higher level of trust, which conflicts with the blockchain paradigm.

Realizations

Verifiable off-chain computations have been proposed and realized in specific blockchain-based protocols and decentralized applications. Still, no framework for generic off-chain computations based on this approach exists.

For example, the Zerocoin [245] and Zerocash [293] protocols leverage zk-SNARK-based, accepting, non-interactive, pessimistic off-chain computations in the context of the specific goal of anonymous payments. However, existing implementations are use case–specific and do not consider or support generic off-chain computations. We provide an in-depth discussion of other related approaches in Section 3.2.

In Part III of this thesis, we demonstrate how generic verifiable off-chain computations, including program specification, execution, and on-chain verification, can be realized in their entirety and supported by a framework

As previously noted, the choice of a verifiable computation scheme is the main determinant of a realization's properties. We provide an in-depth overview of state-of-the-art verifiable computation schemes from literature in Section 2.2.2. Subsequently, we contrast three distinct schemes, each representative for a category of schemes in our overview, with respect to the soft requirements defined above. This analysis allows us to judge the viability of instantiations of the verifiable off-chain computations approach with different schemes.

zk-SNARKs: zk-SNARKs allow a prover to convince a verifier that it executed a program correctly in one message while guaranteeing zero-knowledge. Provable computations are specified as arithmetic circuits [268] or rank-1 constraint system (R1CS) [38]. We introduce these abstractions in detail in Chapter 7.

Proofs are of small constant size (e.g., 127 bytes for the Groth16 [164] scheme over the BN254 curve [27], for example).[2] The complexity of proof generation is quasilinear in the complexity of the off-chain program[3]. This step represents the bottleneck with regards to practicality (see our experimental results in Chapter 11). The verification of proofs is cheap, and its complexity is independent of the complexity of the computation to be checked. This theoretically allows the on-chain verification of arbitrarily complex computations despite the existence of a block complexity limit.

As discussed in Section 2.2.1, zk-SNARKs require a one-time trusted preprocessing step to be performed. Depending on the scheme, this trusted setup can be computation-specific or universal, i.e., usable for all programs (below a complexity limit). To reduce the trust requirements, multi-party computation protocols have been proposed, which distribute the setup over a set of nodes and are secure as long as there is at least one honest participant [36, 58, 59].

Bulletproofs: As described in Section 2.2.2, Bulletproofs [69] are a non-interactive zero-knowledge verifiable computation scheme that does not require a trusted setup. They can be used to prove satisfiability of arbitrary arithmetic circuits or R1CS. The size of Bulletproofs is logarithmic in the complexity of the off-chain program and in the range of a few kilobytes for reasonably small programs. Proof generation complexity is linear in the off-chain program's complexity. However, unlike with zk-SNARKs, the complexity of proof verification also grows linearly with the program's complexity. Thus, on-chain verification for complex off-chain programs can quickly become prohibitively expensive. Batch verification can improve per-proof verification cost.

[2] In this chapter, we only consider zk-SNARK schemes that rely on a trusted setup and achieve constant-size proofs. See Chapter 2 for an in-depth discussion of schemes and their properties.

[3] The off-chain program's complexity is defined through the number of multiplication gates in the representing arithmetic circuits or the number of constraints in the representing R1CS, respectively. In-depth information on this matter can be found in Chapter 7.

zk-STARKs: Like Bulletproofs, zk-STARKs [33] are a non-interactive zero-knowledge verifiable computation scheme that does not require a trusted setup. They use a different arithmetization, the algebraic intermediate representation (AIR), where computational statements are expressed as higher degree polynomials. Nevertheless, the fundamental approach and expressiveness are similar, and AIR programs can be expanded to arithmetic circuits, which enables direct comparison with other verifiable computation schemes.

The size of a zk-STARK-proof is polylogarithmic[4] in the complexity of the associated off-chain program.[5] Proofs range within a few hundred kilobytes for reasonably small off-chain programs. Thus, due to the immense proof sizes, practicality for generic verifiable off-chain computations is limited and only materializes in the case of batched proving of executions of one program. The complexity of proof generation is quasi-linear in the off-chain program's complexity, whereas verification complexity is polylogarithmic.

5.2.2 Enclave-based Off-chain Computations

In this section, we introduce Enclave-based Off-chain Computation which rely on trusted execution environments (TEEs) to realize off-chain computations. The enclave-based off-chain computation approach is accepting, non-interactive, and usually pessimistic.

Concept and Architecture

TEEs, or secure enclaves, are execution environments that guarantee correct code execution as well as confidentiality of data and code through hardware-level isolation.[6] A secure enclave is provided by trusted hardware that protects a dedicated and encrypted address space from unauthorized access and isolates computations in the enclave from outside processes. TEEs are supported by a variety of hardware platforms, for example Intel SGX [10, 90, 176, 238], ARM TrustZone [7], and RISC-V Keystone [219]. TEEs can be combined with remote attestation protocols to realize secure remote computations. A remote attestation protocol allows a client to verify that it talks to an authentic secure enclave on trusted hardware, and that this enclave executes the right program. Note, however, that these protocols require trust in the hardware manufacturer. For an in-depth discussion of remote attestation approaches and how they can be realized, refer to [90].

This secure remote computation approach can be leveraged to realize off-chain computations and implies the architecture depicted in Figure 5.4. To initiate a computation, a client connects to a secure enclave that runs on an off-chain node. This can be either infrastructure the client controls herself or infrastructure owned by a third party; trust assumptions are not affected. The client requests the secure enclave to load the desired off-chain program. Upon completing this initialization step, the enclave provides the client with an attestation that allows her to check the enclave runs on trusted hardware and loaded the right program. This check relies on certificates

[4] Polylogarithmic asymptotic complexity $O(\text{polylog}(n))$ is defined as $O((\log n)^k)$ for some k.

[5] To ensure comparability, the off-chain program's complexity is measured in multiplication gates in the arithmetic circuits resulting from expanding the AIR.

[6] We use the terms secure enclave and TEE synonymously. There is no uniform definition in literature [292].

provided by the trusted hardware manufacturer, which the client has to trust. After successfully completing the check, the client provides public and private inputs to the enclave through an encrypted channel. On completion of the off-chain computation in the secure enclave, the client (which could simply be an unprotected program running on the off-chain node) obtains the result and publishes it to the blockchain. For this result to be trustworthy, the blockchain needs to be sure that it originated from a secure enclave running the correct off-chain program. Thus, an attestation is generated by the secure enclave and forwarded to the blockchain for verification with the computation's result by the client. The on-chain verification of this attestation relies on certificates from the trusted hardware manufacturer, again.

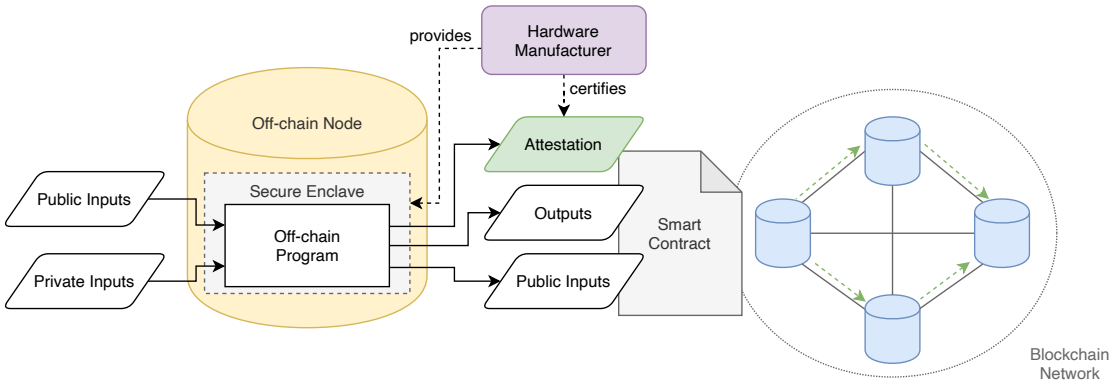


Figure 5.4: Schematic Overview of the Enclave-based Off-chain Computation Approach

Realizations

Enigma [367] and Ekiden [83] present two different realizations of enclave-based off-chain computations. In the Enigma system, programs can either be executed on-chain or in enclaves that are distributed across a separate off-chain network. An Engima-specific scripting language allows developers to mark objects as private to enforce off-chain computations. In contrast to Enigma, Ekiden does not allow on-chain computations, but instead, the blockchain is solely used as persistent state storage. Code and private inputs are provided by an off-chain client that exclusively communicates with enclaves. Once an off-chain computation completes, the enclave returns the output directly to the client while the new state is checkpointed to the blockchain. While optimistic on-chain verification of the attestation would be conceptually possible, it has not been used by either realization.

5.2.3 Incentive-driven Off-chain Computations

In this section, we describe incentive-driven off-chain computations, an approach that leverages economic incentives to guarantee correctness of off-chain computation protocols. For that, it borrows concepts from game theory and mechanism design, in particular [179]. Incentive-driven off-chain computation protocols are optimistic, rejecting, and interactive.

Concept and Architecture

The fundamental assumption in this category of approaches is that participants are economically rational, i.e., behave in a way that maximizes their economic utility. This allows protocols to economically enforce or discourage behavior, for example, by retaining deposits as leverage against malicious activity and financially rewarding desired positive contributions. (Public) blockchains make it particularly easy to incorporate economic consequences into protocols by tying events to native cryptocurrency payments.

The core idea behind incentive-driven off-chain computations is to encourage protocol participants to publish correct off-chain computation results and to punish them for lying, i.e., publishing wrong results. A set of participants independently executes an off-chain program for a set of known public inputs. A participant can publish an initial result claim at any time. Disagreeing participants can then challenge claims by providing alternative, conflicting ones. In case of challenges, the blockchain acts as a judge that settles the dispute and rejects the invalid claim. However, this does not automatically imply correctness of the remaining one. Incorrect computation output claims have to be challenged until the correct result emerges.

To guarantee the emergence of the correct result, an incentive-based off-chain computation protocol must ensure two things:

1. The blockchain can reliably determine and reject an invalid claim in case of disputes.
2. Invalid claims are always challenged; i.e., there exists an active set of challengers.

The first point can be addressed through execution traces, which allow the blockchain to detect a deviation in the off-chain processing steps. By running the deviating step on-chain, the blockchain can reliably judge which one of two conflicting claims is certainly incorrect. The second point can be addressed through mechanism design: Incentives are laid out in a way that incorrect output claims have negative economic consequences and that successful challengers are rewarded.

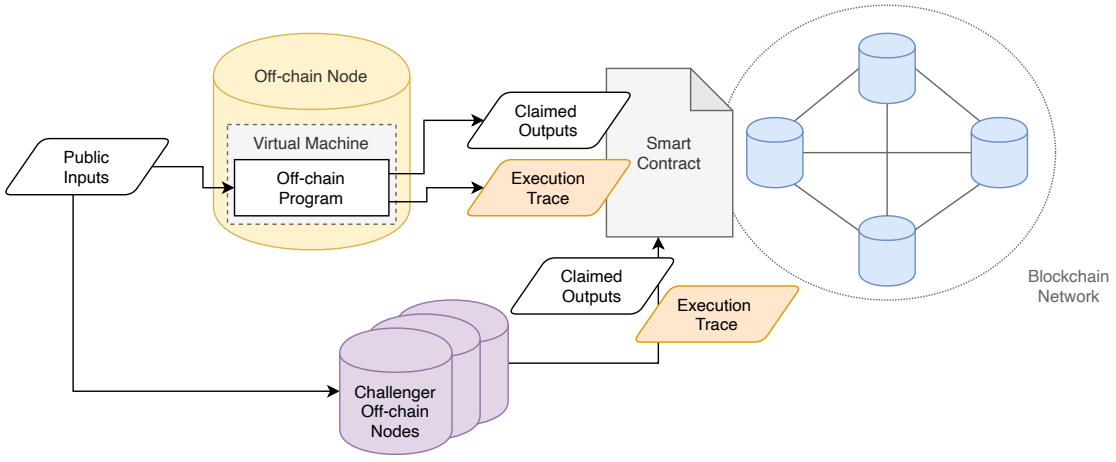


Figure 5.5: Schematic Overview of the Incentive-driven Off-chain Computation Approach

We depict the architecture shared by realizations of the incentive-driven off-chain computation approach in Figure 5.5. The conceptual description above translates to the components in this architecture as follows: A set of off-chain nodes redundantly execute a given off-chain program in a virtual machine that generates an execution trace in the process. Programs have to be specified in a way that allows execution in that virtual machine. The first node to complete the computation publishes the outputs and the associated execution trace to a smart contract, where both are stored. In case any other off-chain node comes to a different result, they challenge the initial claim by sending a conflicting result and the corresponding execution trace to the blockchain. Considering the execution traces, the blockchain acts as a judge and rejects the outputs of the execution that made the (first) processing error.

In actual realizations, on-chain execution traces can be replaced with commitments to reduce the storage footprint. Furthermore, comparing and deciding on the correctness of an off-chain execution traces can be (prohibitively) expensive. Thus, realizations of this approach have to find intelligent ways of reducing the on-chain complexity of the judging process to stay within the blockchain's processing capacity and control costs. More importantly, however, the need for on-chain judgments can and should be minimized through sensible incentive design.

Realizations

As previously explained, realizations of the incentive-driven off-chain computation approach have to address two critical challenges: (1) Keep protocol participants motivated to validate and challenge solutions and (2) reduce computational effort for the on-chain judge.

TrueBit [320] realizes a public marketplace for off-chain computations through an incentive-based design, as previously introduced. The proposal distinguishes between verifiers, who accept off-chain computation tasks and publish results and verifiers, who challenge invalid claims after independent verification. Solvers are directly compensated for their effort by task issuers. As verifiers would stop validating if solvers — as incentivized — only published correct solutions, TrueBit enforces solvers to provide erroneous solutions from time to time and offers a reward to the verifiers for finding them. This ensures that there exists an economic incentive to operate verifiers and that they are available to fulfill their role in the protocol at all times. The problem of the computationally limited on-chain judge is solved through an interactive *verification game* [187, 287] that is played between verifiers and challenging solvers. Applying binary search on the conflicting execution histories, the dispute that the judge must decide on narrows down to a single computational step than can be executed on-chain at low cost.

Arbitrum [193] uses a similar interactive protocol to settle disputes between protocol participants that disagree on off-chain computation results. However, where Truebit uses a global set of verifiers, Arbitrum introduces *managers* that are responsible for the execution of an off-chain program: Before an off-chain program is executed, a set of managers is appointed. Then, the off-chain program is executed by the Arbitrum virtual machine, and managers agree on the result in an off-chain consensus process acceptance through signatures. In case of unanimous consensus of the managers, the smart contract accepts the outputs without further checks. In case of disagreement, the disputed virtual machine processing step is executed on-chain. While

Truebit tries to achieve absolute correctness through its global verifier set, Arbitrum accepts incorrect results in case of unanimous consensus of managers. Thus, parties with interest in a given off-chain computation's result need to become or trust one of its managers. Inputs to an off-chain computation can not be completely private; they have to be shared among managers (which could leak them further). Still, unlike with Truebit, the protocol does not require all inputs to an off-chain program to be published. In case of disputes, the blockchain learns part of the off-chain computation's internal state.

5.2.4 Secure Multi-party Computation-based Off-chain Computations

Secure multi-party computation (MPC) protocols can be leveraged to design a privacy-preserving off-chain computation approach, which we describe in this section. MPC-based off-chain computation protocols are non-interactive, pessimistic, and accepting.

Concept and Architecture

MPC protocols [155, 306, 358] enable a set of nodes to jointly compute a public function on their private inputs in a way that none of the nodes ever has access to any but their own secret inputs. In an active adversary model, MPC protocols cannot guarantee safety and liveness at the same time without an honest majority assumption for the cluster [227]. Oftentimes, protocols require even stronger bounds on the tolerable share of corrupted nodes.

The fundamental idea behind MPC-based off-chain computations is to delegate the execution of an off-chain computation to a cluster of nodes that jointly execute an off-chain program on private inputs as a secure multi-party computation. Through the use of secret-sharing protocols [305], no single cluster node learns the private inputs but only receives an encoded share. This encoded share does not reveal any information about the initial private inputs but can be used in computations. Private inputs can only be recovered from a number of shares larger than a protocol-dependent threshold. Consequently, an attacker would need to control a number of cluster nodes higher than this threshold to extract secret information.

The architecture depicted in Figure 5.6 describes the components involved in realizations of the previously described conceptual idea.

First, the off-chain program to be executed has to be specified as an MPC-compliant circuit and deployed to the off-chain MPC cluster.[7] Then, private inputs are split into shares and the shares are distributed to the cluster nodes as well. Public inputs can be passed to the cluster nodes without splitting. The off-chain MPC cluster nodes jointly execute the off-chain computation. Afterward, at least one node publishes the computation's outputs, including an audit trail.

MPC protocols are commonly designed for settings without active adversaries and prioritize strong confidentiality guarantees over liveness. The specific setting of off-chain computations requires certain non-standard properties to be fulfilled by MPC protocols employed in this context:

[7] MPC circuits can be boolean or arithmetic depending on the type of protocol. While garbled circuit protocols [358] assume boolean circuits, secret sharing-based protocols [155] rely on arithmetic circuits.

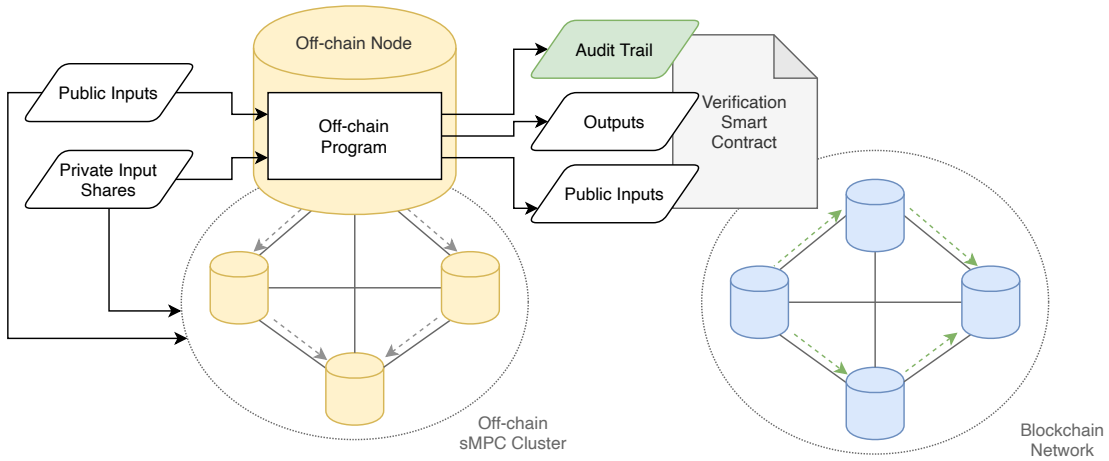


Figure 5.6: Schematic Overview of the MPC-based Off-chain Computation Approach

Guaranteed Output Delivery: A protocol with guaranteed output delivery ensures that all participating parties receive output [87]. Delivery cannot be prevented by adversaries. Hence, guaranteed output delivery ensures liveness in case of failures [24], which have to be expected in the off-chain computation setting.

Public Auditability: As mentioned above, the set of MPC protocols with suitable liveness and safety properties for our context rely on an honest majority assumption. Under this assumption, all nodes could publish their outputs to the blockchain on protocol completion, which would then accept the output reported by the majority as the correct computation result. However, violations of the honest majority assumptions are not detectable. A dishonest majority could publish incorrect results to the blockchain. In the context of off-chain computations, however, we have to guarantee correctness in all cases and do not want to trust that this assumption holds. Thus, suitable MPC protocols should be publicly auditable [29], i.e., an *auditor* not involved during the protocol can check the correctness of the computation. This check is reliable even if all parties participating in the MPC are corrupted. Given public auditability, correctness can be checked by an on-chain auditor, i.e., the verification smart contract, or an arbitrary off-chain auditor by evaluating a computation's on-chain audit trail after outputs have been optimistically accepted.

Efficiency: Not only the multi-party computation itself but also the on-chain verification of an audit trail has to be practically efficient. This requirement implies that audit trails have to be small.

5.2.5 Realizations

Generic MPC-based off-chain computations have not been successfully realized beyond research prototypes [366].

In the original Enigma paper [367], Zyskind proposed a privacy-preserving decentralized com-

putation platform based on a publicly auditable secret sharing MPCs scheme [29]. However, presumably because of the protocol's considerable performance overhead, the Enigma project chose to postpone MPCs-based realization and instead relies on TEEs as already explained in Section 5.2.2 [116]. Related theoretical approaches based on fully homomorphic encryption (FHE) [152] involve even higher overheads and are not efficient enough to be practical [257].

Another line of work ties secure multi-party computations to blockchain transactions to improve fairness properties of MPCs by penalizing abortion and enable protocols with complex conditional cash redistribution, e.g., poker games, auctions and lotteries [14, 41, 213]. While related, these proposals pursue different goals and involve the blockchain in several protocol stages. Hence, they cannot be considered realizations of the MPC-based off-chain computation approach.

Recently, with HoneyBadgerMPC [227] and Blinder [1], two novel practical and scalable MPC constructions that guarantee output delivery have been proposed. However, these protocols are not publicly auditable and can thus only be used in settings where an honest majority can be guaranteed.

5.3 Comparative Assessment

Finally, we contrast the different approaches to off-chain computations we identified in a comparative assessment and discuss their specific properties.

A detailed overview with regards to the properties of different off-chaining approaches is given in Table 5.1. We elaborate on and compare notable aspects in more detail in the remainder of this section.

Table 5.1: Comparison of Off-chain Computation Approaches

		Scalability		*Privacy*	*Security*		*Programmability*
Approach	**Realization**	**On-chain Verification**	**Off-chain Computation**	**Private Inputs**	**Security Assumption**	**Post-Quantum Security**	**Progr. Abstraction**
Verifiable Computation-based	zk-SNARKs [164, 166, 268]	*One-time setup:* $O(n)$, n number of multiplication gates in circuit *Repeated verify step:* $O(1)$ *Proof size:* $O(1)$, e.g., 3 group elements [164], i.e., 127 bytes for BN254 curve [27]	$O(n \log n)$, n number of multiplication gates in circuit	yes	Knowledge of exponent assumption [151] & trusted setup was performed correctly	no	Arithmetic circuits
	Bulletproofs [69]	*Verify:* $O(n)$, n number of multiplication gates in circuit *Proof size:* few kilobytes, $O(\log n)$, n number of multiplication gates in circuit	$O(n)$, n number of multiplication gates in circuit	yes	Discrete log problem	no	Arithmetic circuits
	zk-STARKs [33]	*Verify:* $O(\log^2 n)$, n number of multiplication gates of AIR expanded to circuit *Proof size:* few hundred kilobytes, $O(\log^2 n)$, n number of multiplication gates of AIR expanded to circuit	$O(n \log^2 n)$, n number of multiplication gates of AIR expanded to circuit	yes	Collision-resistant hash functions	yes	AIR using higher degree polynomials (expandable to circuits)
Enclave-based		Validate enclave's attestation: $O(1)$, singature verification [90]	Native execution & attestation overhead	yes	TEEs are isolated & trust in remote attestation certificates	no	Languages that compile into machine code executable by TEE
Incentive-based		Binary search & one computation step: $O(\log n)$, n number of computation steps [320]	Virtual machine overhead (execution history)	no	Economically rational participants	yes	Languages that compile into VM instruction set used on- and off-chain
MPC-based		*On-chain Auditor:* $O(n)$, n number of gates in circuit [29] *Audit Trail Size:* $O(n)$, n number of gates in circuit [29]	$O(n)$, n number of gates in circuit	yes	At least one honest node & honesty threshold for private input protection and liveness	yes	Boolean or arithmetic circuits

5.3.1 Scalability

To increase the computational throughput that can be processed by the blockchain, the on-chain verification of an off-chain computation needs to be as cheap as possible. Additionally, the execution of off-chain computation needs to remain reasonably fast and inexpensive to be considered practical.

Due to their non-interactive, pessimistic, and accepting nature, realizations of the verifiable off-chain computation approach finalize results in only one on-chain transaction. For zk-SNARKs, proof size and on-chain verification complexity are extremely small and independent of circuit complexity, making them an ideal candidate for off-chain computations. Like in all verifiable computation schemes, the proof generation step, which is part of the execution of a verifiable off-chain computation, does involve significant overhead compared to native execution on standard processors. However, as we demonstrate in Chapter 11 and Part IV, this overhead is sufficiently small for the scheme to be practical for useful off-chain programs. For zk-STARKs and Bulletproofs, proof size and verification complexity grow with the off-chain program's complexity, limiting applicability. Proof sizes for non-trivial off-chain computations are too big to be processed by state-of-the-art public blockchains like Ethereum.

From a scalability perspective, secure enclave–based approaches are remarkably efficient. The blockchain only needs to perform a simple signature check, and the execution of an off-chain computation in a TEE involves very little overhead compared to native execution.

Due to their interactive, rejecting, and optimistic nature, incentive-based off-chain computations require a dispute period and potentially several on-chain transactions before an off-chain computation is finalized. Using advanced dispute resolution protocols as introduced by this approach's realizations, on-chain verification becomes feasible and inexpensive. Off-chain program execution in suitable virtual machines does involve some overhead but does not significantly impair practicality [193, 320]. Redundant off-chain execution is a direct consequence of the incentive-based design.

We are not aware of a protocol with the right combination of properties to instantiate MPC-based off-chain computation, i.e., guaranteed output delivery and public auditability under realistic honesty threshold assumptions for clusters. Current publicly auditable protocols suffer from large proof sizes and high off-chain computation costs.

5.3.2 Privacy

MPC-based off-chain computations offer strictly stronger privacy guarantees than the other approaches: No single party ever needs to know an off-chain computation's (private) inputs. In the other approaches, all private inputs have to be available to a single off-chain node. Here, shared computations on private data, e.g., auctions or voting, require trust in that off-chain node with regards to privacy.

Incentive-based off-chain computation approaches do not support private inputs. They can not be used as a privacy-engineering approach. All other approaches do support private inputs that remain secret to the node that executes the off-chain computation.

In recent years, there have been several successful attacks on secure enclaves, where secret information could be extracted [63, 82, 248, 297, 329]. This has to be considered in case the off-chain node is not controlled by the party who initiated the computation executed thereon and provided private inputs.

5.3.3 Security

Weak and standard assumptions are better than complex and uncertain ones. Post-quantum security is an aspect that may become critical in the future. While noteworthy, we do not consider it to be a crucial requirement for off-chain computation approaches today.

In the case of verifiable off-chain computations and MPC-based approaches, cryptographic proof[8] is used to convince the blockchain by relying on cryptographic assumptions rather than economic ones or trusted hardware. As previously explained, zk-SNARKs require a trusted setup phase and are based on the relatively novel cryptographic knowledge of exponent assumption [151]. However, since invalidating this assumption would allow access to funds on the Zcash blockchain [361], it can at least be considered battle-hardened by now. The security assumptions for Bulletproofs and zk-STARKs are well-established cryptographic standard assumptions.

While incentive-based systems like TrueBit [320] are safe under the assumption that there will always be at least one rational participant sufficiently motivated by economic incentives, liveness can be impaired by malicious participants: They can challenge every computation step with erroneous solutions and hence enforce full on-chain execution.

Enclave-based approaches require trust in the manufacturer of hardware platform that hosts TEEs, e.g., Intel SGX. These have been shown to be vulnerable [260]. Furthermore, remote attestations requires trust in manufacturer-issued certificates [90, 183].

5.3.4 Programmability

Enclave-based and incentive-based off-chain computations rely on standard computational models. Standard compilation techniques can be and are used to translate high-level programming languages into executable programs. The Truebit virtual machine, for example, relies on WebAssembly (Wasm) [341] as its instruction set and a wide range of high-level programming languages supports compilation to Wasm. Specific software development kits support developers in specifying TEE-compatible executables in established high-level programming languages like C++ or Rust [137, 184].

Verifiable off-chain computation and MPC-based approaches, in contrast, rely on arithmetizations of programs into arithmetic or boolean circuits. This makes them notoriously difficult to program and requires deep expertise concerning specific protocols to ensure efficiency. To make matters worse, different realizations may require different concrete circuit representations. We discuss circuit specification in Sections 3.2.3 and 8.1.

[8] These proofs can be probabilistic, they are also referred to as arguments in that case.

5.4 Conclusion

In this chapter, we set out to find an approach that realizes scalable off-chain computations with strong privacy guarantees. For that purpose, we introduced and compared different off-chain computation approaches and discussed concrete realizations.

We found that incentive-based off-chain computations represent a promising scalability approach. They cannot be used as a privacy-engineering tool, however, since they do not support private inputs for off-chain programs. Unfortunately, none of the proposed realizations have yet been deployed. Since enclave-based off-chain computations require trust in the manufacturer of the hardware that hosts trusted execution environments (TEEs), they do impair blockchains' trustlessness property to a significant and generally unwanted degree. Realizations of MPC-based off-chain computations require MPC protocols that guarantee output and can be publicly audited. Such protocols have yet to be developed. Scalable and robust MPCs are an active area of research and may enable off-chain computation with strong privacy guarantees in the future.

From our analysis, we conclude that zk-SNARK-based verifiable off-chain computations are a particularly promising approach. They realize strong privacy guarantees and can convince the blockchain of a computation's correctness through a small and cheap to verify proof. The trust required in the necessary setup step can be minimized through an open, distributed setup procedure. However, due to their arithmetic circuit based programming model and the complexity of the underlying verifiable computation schemes, they are notoriously hard to instantiate and use in practice.

Off-chain computations need to be accessible to developers to represent a powerful privacy engineering and scaling tool. Yet, there is neither a high-level language for the convenient specification of zk-SNARKs-based verifiable off-chain computations nor are there tools to support developers with the on-chain verification of zk-SNARK proofs. Usability in general and programmability, in particular, represent major hurdles in the adoption of zk-SNARKs-based off-chain computations.

To address this issue, we introduce ZoKrates, a framework supporting zk-SNARKs-based verifiable off-chain computations from program specification to on-chain verification in the next part of this thesis. ZoKrates addresses the aforementioned programmability problem by providing a high-level domain-specific programming language that facilitates the specification of efficient off-chain programs by developers without deep cryptographic expertise. This programming abstraction supports arbitrary programs that can be used in a wide range of use cases. To ensure usability along the whole off-chain computation process, the framework automatically generates verification smart contracts for off-chain programs to hide the complexity of the underlying verifiable computation schemes.

Part III

ZoKrates

From the comparative assessment of off-chain computation approaches in Chapter 5, we concluded that zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) are suitable verifiable computation schemes to enable efficient off-chain computations in zero-knowledge.

However, having identified a suitable class of zero-knowledge verifiable computation schemes is not sufficient to instantiate zk-SNARK-based verifiable off-chain computations as introduced in Section 5.2.1 in practice: The low-level abstractions zkSNARKs operate on are tedious to program and hardly accessible for developers. To address this problem, we propose and introduce ZoKrates, the first language and toolbox for usable and efficient off-chain computations. ZoKrates hides away significant complexity when specifying verifiable off-chain computations by providing a higher-level language that compiles to statements provable with non-interactive succinct zero-knowledge proofs. Furthermore, it provides extensive tooling to support the execution of such off-chain programs as well as their verification on a blockchain. In summary, ZoKrates represents the first comprehensive verifiable off-chain computation framework and is truly accessible to developers without specialist knowledge end-to-end.

After defining design goals, we provide a user-centric ZoKrates overview. In subsequent chapters, we describe ZoKrates in detail: First, we introduce the abstractions underlying zkSNARKs and analyze which computations can be expressed and proved efficiently. Based on these insights, we derive the ZoKrates intermediate representation (ZIR), a low-level abstraction that serves as ZoKrates' compilation target. Afterwards, we design and specify the ZoKrates language and its translation to this intermediate representation. Subsequently, we introduce and discuss ZoKrates' architecture and provide an overview of our implementation thereof. We conclude with an experimental performance evaluation of ZoKrates-based off-chain computations.

This part contains material previously published at IEEE Blockchain 2018 [113]. The ZoKrates implementation is fully open-source and available on GitHub[9]. This implementation uses the Ethereum blockchain for proof verification, but our approach and architecture are blockchain-agnostic. In the implementation at hand, only the generation of on-chain verification logic is specific to Ethereum and can easily be extended to support other platforms.

[9] `https://github.com/ZoKrates/ZoKrates`

CHAPTER 6

Design Goals and Overview

In this chapter, we motivate ZoKrates, describe its goals, and provide a high-level overview from a user's perspective before describing theoretical foundations, the ZoKrates language, the system architecture, and its implementation and evaluation in subsequent chapters.

We define usability, efficiency, and generality as main objectives for ZoKrates as a framework for zero-knowledge verifiable off-chain computations. A framework that achieves these goals constitutes a powerful privacy-engineering and scalability tool.

6.1 Motivation & Design Goals

In previous chapters, we identified verifiable off-chain computations as a promising approach to address privacy concerns and scalability in the context of blockchains. Our analysis in Chapter 5 identified zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) as suitable verifiable computation schemes to instantiate the verifiable off-chain computation pattern described in Section 4.3.2.

However, leveraging this insight in practice is hard. Even for cryptography experts, who are familiar with zk-SNARKs and related abstractions, specifying meaningful provable computations is challenging, time-consuming and error-prone. This complexity puts the direct application of verifiable computation schemes out of reach for blockchain developers. These developers generally program in high-level languages, e.g., Solidity, and require a way of specifying off-chain computations that is compatible and at a similar level of abstraction. In addition to the challenge of program specification, they would have to implement on-chain verifiers; a task that requires in-depth understanding of the employed verifiable computation scheme. As domain experts, blockchain developers are aware of the intricacies of the blockchain platform they develop for, but cannot be concerned with the low-level implementation of complex cryptographic schemes.

In conclusion, there is a gap between the blockchain domain and its developers' needs and

existing verifiable computations schemes and their underlying abstractions (illustrated in Figure 6.1).

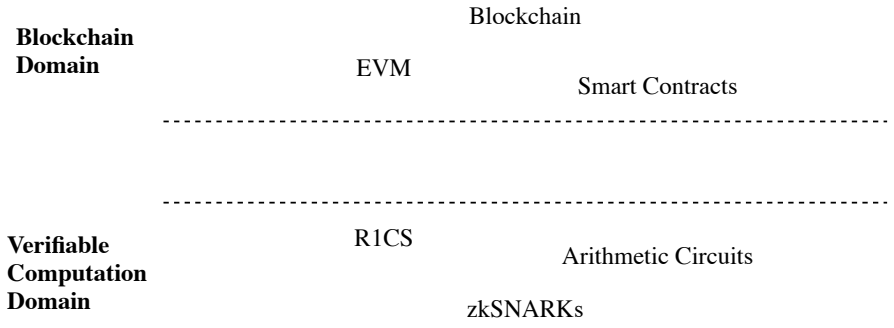


Figure 6.1: Gap between Problem Domains.

We aim to bridge this gap by defining a suitable, domain-specific programming abstraction to specify efficient verifiable off-chain computations for developers and by providing tooling so that program execution, proving, and on-chain verification become simple and do not require any insight into the verifiable computation scheme employed. Hereby, it is crucial to respect the idiosyncrasies of both the blockchain and verifiable computation domains to ensure usability, generality, and efficiency — our primary design goals. Subsequently, we describe these three design goals that our solution tries to achieve in more detail. These high-level goals reflect in the design decisions made and described in later chapters.

6.1.1 Usability

Developers who want to leverage off-chain computations as a privacy-engineering tool or reduce the on-chain footprint of their decentralized applications should be provided with a convenient, high-level abstraction that does not require to be aware of the intricacies of verifiable computation schemes. A library that supports developers with common tasks should be available.

Both, the domain-specific programming language as well as the tools provided for the process of executing, proving, and verifying off-chain computations should have intuitive interfaces that abstract verifiable computation scheme–specific details.

6.1.2 Generality

The domain-specific language should be capable of expressing all computations that are efficiently provable (for a detailed analysis, see Chapter 7). Furthermore, both the language and all tooling should be as agnostic of a specific blockchain and specific verifiable computation schemes as possible.

6.1.3 Efficiency

The efficiency goal refers to the domain-specific language for off-chain computations as well as the tooling provided: Programs written in the domain-specific language should translate to an

efficiently provable low-level representation. This goal refers to the low-level abstraction itself as well as specific program representations in that abstraction. Tools supporting the off-chain computation process, e.g., executing an off-chain program, should have adequate performance.

6.2 ZoKrates Overview

In this section, we provide an overview of ZoKrates, a language and set of tools for efficient and usable verifiable off-chain computations, from a user-centric perspective. In subsequent chapters, we describe the ZoKrates language, the architecture, and its implementation in more detail.

We describe the process a user goes through when specifying, executing, proving, and verifying off-chain computations using ZoKrates. As a guiding example, we use the example of proving knowledge of a hash's pre-image without revealing it in the process. In our introductory overview, a single user performs all steps. That does not have to be the case in actual ZoKrates applications (see Part IV).

In Figure 6.2, we provide an overview of all steps involved from specifying a computation through a ZoKrates program to verifying its execution on the blockchain.

Initially, a set of one-time preparation steps are performed for an off-chain program. These steps are marked with ◯. Steps that are repeated for every execution of that program are marked with ◇.

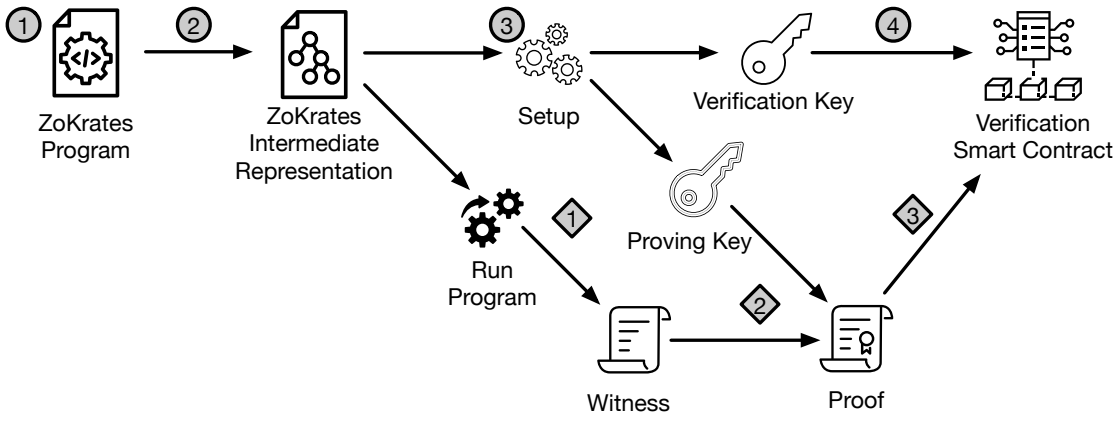


Figure 6.2: Overview of ZoKrates Process Steps

6.2.1 Program Specification

As a first step, a developer specifies a ZoKrates program in the domain-specific ZoKrates language that encodes the off-chain computation ①. The ZoKrates language is a simple imperative language with static scoping designed to be easy to use and, at the same time, translate into a provable abstraction with very little overhead. For an in-depth introduction and specification, refer to Chapter 8.

Listing 6.1 shows a program that computes the SHA-256 hash function of a set of inputs. The main function serves as entry-point and the private keyword preceding the inputs marks them as private, i.e., they remain a secret to the prover and do not become public with the proof attesting correct program execution. The inputs are of the type field, which can be considered unsigned integers that encode 128 bits for the purpose of this example. We provide a detailed explanation of this data type in Section 7.1. As the main function's signature shows, the return statement returns two field elements, which represent the SHA-256 hash. Again, two field elements are used to encode 128 bits each.

Listing 6.1: ZoKrates Program Computing SHA-256-Hash

```
1 import "hashes/sha256/512bitPacked" as sha256packed
2
3 def main(private field a, private field b, private field c, private field d)
      -> (field[2]):
4   h = sha256packed([a, b, c, d])
5   return h
```

Files containing ZoKrates source code have the zok filename extension. For this introductory overview, we assume that the program given in Listing 6.1 is stored in a text-based file with the name sha256-hash.zok. In the remainder of this chapter, we show how a user can prove knowledge of a hash's pre-image based on this program using the ZoKrates command-line interface (CLI).

6.2.2 Compilation

After program specification, the ZoKrates program is compiled ② into the ZoKrates intermediate representation (ZIR), a custom format derived in Chapter 7, which serves as efficiently executable and provable base abstraction for all further steps. To trigger compilation, the user issues the following CLI command:

```
zokrates compile -i sha256-hash.zok
```

As a result, the ZoKrates compiler creates two files:

```
out
out.ztf
```

The out file contains the resulting ZIR in binary format, which is used in further processing steps. out.ztf is a human-readable representation of the ZIR encoded in out in the ZoKrates text format (ZTF) for debugging and analysis (see Section 7.4).

6.2.3 Setup

As explained in Chapter 2, zk-SNARK verifiable computation schemes are preprocessing schemes and require a one-time setup to be performed before proofs can be generated.[1]

The setup is trusted in the sense that whoever executes it needs to forget private information used in the process. If that information is not forgotten, the owner could potentially create fake proofs. However, the zero-knowledge property of other proofs would not be impaired.

ZoKrates offers support for local trusted setups. Based on the previously generated ZIR, the setup step generates two keys, a long proving key and a short verification key ④. These keys are required to create and verify proofs, respectively, and can be reused an arbitrary number of times. Both keys are public information, i.e., they do not have to be kept confidential. Depending on the chosen verifiable computation scheme, the keys can be program-specific or universal, i.e., applicable for arbitrary computations (see Section 2.2.1).

ZoKrates' local setup should be used, during development or by a trusted party if available. In the context of public, untrusted settings, secure multiparty-computation protocols and implementations are available [36, 58], which can replace the local trusted setup.

A user runs the local setup step for a compiled program through the following command, which uses the `out` file by default if existent:

```
zokrates setup
```

The following two files containing the public keys for proving and verifying computations are generated:

```
proving.key
verification.key
```

By default, the Groth16 [164] proving scheme is employed; other schemes can be selected through a CLI option. Note that ZoKrates can, by design, seamlessly support verifiable computation schemes that do not require a trusted setup. However, these schemes are not practical yet for our use case.

6.2.4 Verification Contract Generation & Deployment

To ensure correctness of off-chain computations, an on-chain verification component is required: A proof submitted to the blockchain is decided correct or invalid by this component by performing a set of verification steps defined by the employed verifiable computation scheme.

ZoKrates supports developers with this process without exposing them to the intricacies of a scheme's verification logic. It provides tools for the convenient generation of verification smart contracts for given ZoKrates programs ③.

[1] In this chapter, we only consider zk-SNARK schemes that rely on a trusted setup and achieve constant-size proofs. See Section 2.2 for an in-depth discussion of schemes and their properties.

In the context of the `SHA-256` hashing example, the user issues the following CLI command to generate a verification smart contract:

```
zokrates export-verifier --abi v2
```

In response, ZoKrates generates a Solidity Smart Contract in `verifier.sol` (using Solidity ABI version 2), which contains the previously generated verification key. We depict an excerpt of relevant parts from this contract in Listing 6.2.

Listing 6.2: Solidity Verification Smart Contract for `SHA-256` Program

```
contract Verifier {

    struct VerifyingKey {
      ...
    }
    struct Proof {
      ...
    }

    event Verified(string s);

    function verifyTx(
            Proof memory proof,
            uint[2] memory input
        ) public returns (bool r) {
        uint[] memory inputValues = new uint[](input.length);
        for(uint i = 0; i < input.length; i++){
            inputValues[i] = input[i];
        }
        if (verify(inputValues, proof) == 0) {
            emit Verified("Transaction successfully verified.");
            return true;
        } else {
            return false;
        }
    }
}
```

The `verifyTx` function receives proofs for verification. Besides the proof itself, it accepts two inputs, which represents the two field elements returned by the ZoKrates `SHA-256` program (see Listing 6.1). We explain this in more detail in Section 6.2.7.

In the existing ZoKrates implementation, the generated verification contract is specific to the Ethereum blockchain. However, this is rather an implementation detail than a conceptual limitation. As described in Chapter 2, the verification step consists of a relatively simple set of cryptographic operations on an elliptic curve. Hence, the generation of on-chain verifiers for other blockchains or off-chain verifiers is a straight-forward process.

After generation, the verification smart contract needs to be deployed. ZoKrates does not offer

support for this step, as contract deployment is a standard task supported by APIs and development tools for blockchains, e.g., the web3 API [128] or Truffle [325].

The verification contract generation and its deployment complete the initial set of one-time preparation steps that need to be performed executing and verifying off-chain executions of a ZoKrates program.

6.2.5 Off-chain Program Execution

In previous sections, we described how an off-chain program can be specified using the ZoKrates language as well as its compilation to the ZIR. We performed these steps and a trusted setup for the `SHA-256` program given in Listing 6.1. Furthermore, we prepared a verification smart contract, which checks proofs attesting execution correctness for that off-chain program.

Before such a proof can be generated, the program has to be run. More precisely, the program is interpreted by the ZoKrates interpreter ⟨1⟩. During execution, the interpreter computes a witness, i.e., values for input, output, and intermediate variables that satisfy the constraints defined in the ZIR. Such a witness can be interpreted as an execution trace and enables checking that every computational step has been performed correctly post execution. Refer to Chapter 7 for an in-depth description and discussion of the concept.

As previously explained, the `SHA-256` example program calculates a hash of 512 input bits encoded in four field elements. When running the program, values for these inputs need to be provided as arguments. For simplicity, we set these arguments to 0^{512} in this example. The following command executes the compiled program (encoded in the `out` file that serves as default input) with the specified arguments:

```
zokrates compute-witness --arguments 0 0 0 0
```

The witness derived in the process is stored as a set of variable-value pairs in a file named `witness` by default.

Execution of the `SHA-256` example program by the ZoKrates interpreter yields the following result, where both entries are field element that encode 128 bits of the 256 bit hash:

```
["326522724692461750427768532537390503835","89059515727727869117346995944635890507"]
```

Converting these field elements to one result string in hexadecimal notation, which is more common for `SHA-256` hashes, gives:

```
"0xf5a5fd42d16a20302798ef6ed309979b43003d2320d9f0e8ea9831a92759fb4b"
```

6.2.6 Proving Correct Execution

After a proving key has been generated and a witness has been computed by running a program, a proof attesting the correctness of that program's execution, i.e., a proof proving the witness's validity, can be created ⟨2⟩.

For a given witness (expected in file `witness` unless explicitly specified) and a proving key (file `proving.key` by default), the following command generates a proof and stores it in the `proof.json` file:

```
zokrates generate-proof
```

The resulting proof is very small (see Chapter 11 for details) and can thus be efficiently sent to the blockchain via a network. Listing 6.3 shows this proof, which was generated using the default Groth16 proving scheme. It consists of three elliptic curve points a,b,c, which serve as inputs to the verifier. Other verifiable computation schemes available for selection have a slightly different proof structure.

Listing 6.3: Proof and Inputs in JSON Format

```
{
    "proof": {
        "a": [
            "0x20c06ac8fdde96d140b434569283d10b52fe4d0a6fe636360f59c5ad3f99c2fe",
            "0x14350e443a0b79566b833965b7a2dd87a2d8259dd6d65e7d7d20a8bf7fb0e392"
        ],
        "b": [
            [
                "0x18f5bc0808d2949d270e01a4a7dd930525d9b1058957d99eeb4ed7159ea1b5ff",
                "0x0062f8290f045fb984b0877fc8fd3fddd4e5143d25c16c0104a8f890b639c9b2"
            ],
            [
                "0x18a252f0e55cbade6586317f97a2dd966b16b6a544869a59cb9cb504910aa1dd",
                "0x2cf512e4566855d63dc9f1073c77323a634b006f05702510f9fd8939a1012262"
            ]
        ],
        "c": [
            "0x1a49b6cd8b4de514180a930ae86664820400efa3cb2a51d0f36a10e357330f93",
            "0x16f344aa618648d1acb8345ac4e50d67d0c69a9127dc0f358516c2f9d2772766"
        ]
    },
    "inputs": [
        "0x000000000000000000000000000000000f5a5fd42d16a20302798ef6ed309979b",
        "0x0000000000000000000000000000000043003d2320d9f0e8ea9831a92759fb4b"
    ]
}
```

Besides the proof, Listing 6.3 also contains an input object. Comparing the input object's values to the `SHA-256` computation results in Section 6.2.5, it can be seen that these values equal the computation results in hexadecimal encoding. It may sound counterintuitive that an input field contains computation results, i.e., outputs of the initial ZoKrates program. However taking a verifier's perspective, who receives proofs and potentially additional information as input, clarifies this choice of name: The information contained in Listing 6.3 is sent to a verification smart contract. Thus, the input object comprises all information that should become public with the proof. This includes outputs and public inputs of the ZoKrates program the proof was generated for.

6.2.7 On-chain Proof Verification

After proof-generation, which — if valid — certifies that an off-chain computation has been performed correctly, the resulting proof is sent to the verification smart contract ③. This on-chain component then performs the verification logic as encoded in the contract source code (see Listing 6.2) and accepts or rejects.

Like in the case of deployment of verification smart contracts, ZoKrates does not directly support issuing verification transactions to the blockchain. Several standard tools already address this need.

ZoKrates does, however, support developers in interacting with these tools by providing proofs in the required formats, e.g., through the following command:

```
zokrates print-proof --format remix
```

This command prints the proof in a format compatible with the Web-based Remix IDE[2] depicted in Figure 6.3 that supports ZoKrates development. That way, a proof can directly be sent to a verification smart contract from within the tool.

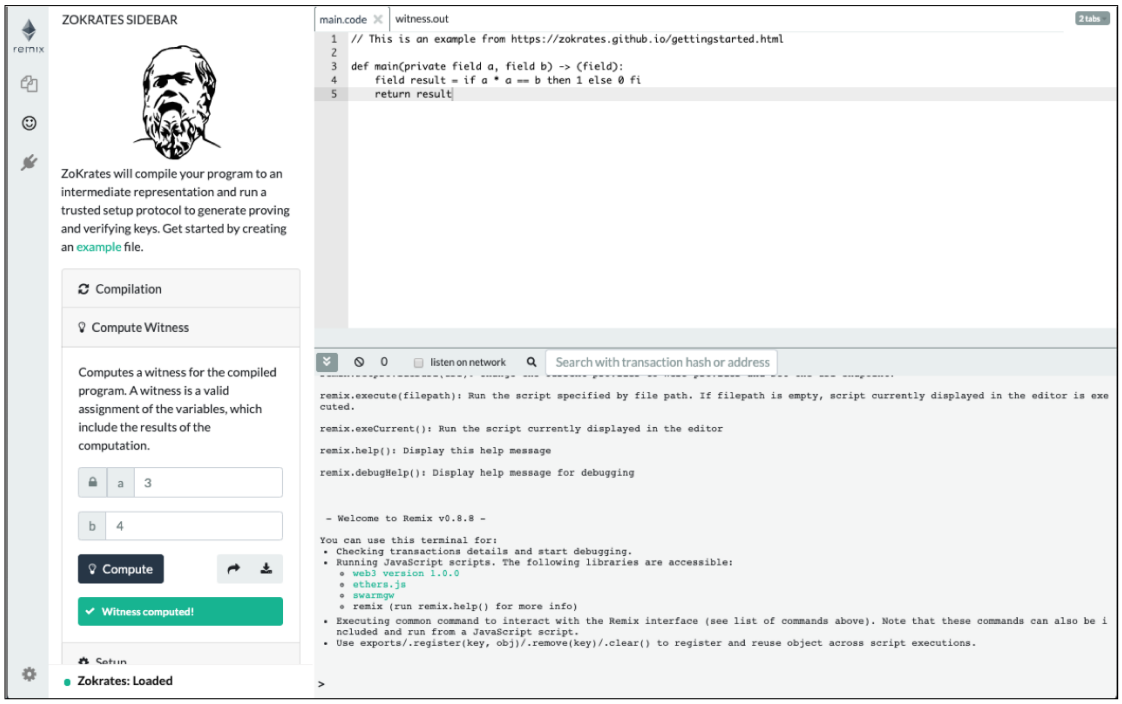


Figure 6.3: ZoKrates Development in the Remix IDE

[2] https://remix.ethereum.org/

CHAPTER 7

ZoKrates Intermediate Representation

In this chapter, we motivate and derive the ZoKrates intermediate representation (ZIR), an executable and provable low-level abstraction that serves as the compilation target for the ZoKrates language. We lay the foundation for the derivation of this format by providing an overview of the mathematical foundations and representations involved in specifying programs provable with zk-SNARKs in Sections 7.1 to 7.2. Readers familiar with prime fields, arithmetic circuits and R1CS can skip these sections. We discuss the theoretical expressiveness of these abstractions as well as their limitations in Section 7.3. Based on these findings, we motivate the need for a new abstraction that supports efficient execution and proving. As a result of these considerations, we introduce the ZIR as a program representation and format that satisfies our requirements.

Specifying programs so they can be proved with zk-SNARKs is quite different from specifying programs for regular instruction set architectures implemented in hardware. Physical processors have access to conditional jumps, memory, rich data types, recursion, and operations on binary data in order to derive results. Compilers target instruction set architectures that use this rich set of features and allow Turing-complete programs to be executed for arbitrary inputs.

In contrast to this, abstractions for verifiable computations, in particular zk-SNARKs, are far more primitive and less powerful. As described in Chapter 2, zk-SNARK protocols [151, 164, 166, 268] can be used to prove and verify satisfiability of arithmetic circuits or R1CSs over a prime field $\mathbb{F}_p$. To lay the foundation for our contributions, we introduce these abstractions and establish the non-obvious link between algebraic structures and program execution.

Building on that, we analyze the expressiveness of the abstractions and discuss their limitations and how these can be addressed to achieve practical verifiable computations. Condensing these insights, we introduce the ZIR, an abstraction that serves as the compilation target for ZoKrates programs. It is designed to support both, efficient program execution as well as proof generation.

As the goal is to develop an intuition and describe relationships between commonly used abstractions, we omit a rigorous formal treatment here and refer the interested reader to the references provided.

For the analysis conducted in this chapter, it is sufficient to consider zk-SNARKs as a black box mechanism that can be used to succinctly prove circuit satisfiability or validity of a variable assignment for a R1CS in zero-knowledge. Refer to Chapter 2 for more information on their inner workings and properties.

7.1 Prime Fields and Arithmetics

On the lowest level, zk-SNARKs can be used to prove and verify the correctness of arithmetic operations in a prime field. In this section, we introduce this fundamental algebraic structure. This introduction lays the foundation for subsequent sections, where we describe the organization of arithmetic operations over prime fields into provable abstractions.

Prime Field Definition: A field is a set of elements for which two basic arithmetic operations, addition and multiplication, with the following properties are defined: The operations are commutative, associative, and multiplication distributes over addition. Furthermore, for every element, there exists an additive inverse, and there exists a multiplicative inverse for all elements but 0, i.e., the neutral element of addition.

For example, the real numbers $\mathbb{R}$ with addition and multiplication and their inverse operations, subtraction and division, represent a field.

In the context of zk-SNARKs, the arithmetic operations are not defined over $\mathbb{R}$, but over a finite set of numbers defined by the following rule:

$$x \equiv z \pmod{p}$$

More specifically, since p is prime, this finite set has a prime number of elements. Intuitively, one can think of such a set as the set of natural numbers smaller than p. For this set, we define the following two basic arithmetic operations, addition and multiplication as follows.

$$\begin{aligned} a + b &= c \pmod{p}, \quad a, b \in \mathbb{F}_p \\ a * b &= c \pmod{p}, \quad a, b \in \mathbb{F}_p \end{aligned}$$

These operations are invertible, commutative and associative and together fulfill the distributive law. Hence, this structure fulfills the field criteria, and we define the prime field:

$$\mathbb{F}_p = \mathbb{Z} \qquad \pmod{p}$$

In summary, prime field elements can be added and multiplied as in real number arithmetics; afterward, the $\pmod{p}$ operation needs to be applied. Software developers are familiar with this

concept of modular arithmetics from overflows in other data types, e.g., adding to an unsigned integer so that the result exceeds the data type's maximum value. In this case, the result is truncated back to the data type's bitwidth, which can be expressed through the modulo operator, as shown in the following example:

$$max\_value(uint32) + 1 = (2^{32} - 1) + 1 = 2^{32} \pmod{2^{32}} = 0$$

By definition, there exists an additive inverse for all field elements and there exists a multiplicative inverse for all field elements but 0. We define subtraction and division as addition and multiplication with these inverse elements as follows:

$$\begin{aligned} a - b &= a + (-b) \pmod{p}, \quad a, b \in \mathbb{F}_p \\ a/b &= a * (b^{-1}) \pmod{p}, \quad a \in \mathbb{F}_p,\, b \in \mathbb{F}_p \setminus \{0\} \end{aligned}$$

Note that divisions may behave differently than expected. The results coincide with those of regular integer division, as commonly encountered in high-level programming languages [220], for cases where $a \bmod b = 0$. They differ for $a \bmod b \neq 0$, however, since the operation does not truncate [223]. The multiplicative inverse of a, a^{-1}, can be efficiently computed using the Extended Euclidean Algorithm [195].

Prime Field Example: We illustrate a prime field for $p = 13$ in Figure 7.1. Projecting the number line on a circle captures the notion of "breaking back" results of operations outside of the interval $[0, p-1]$ into the field thought the $\bmod$ operation.

Addition and multiplication can be visualized as clock-wise movement on the prime-field circle, whereas subtraction and division move the result-pointer counter-clockwise.

Applying the arithmetic rules defined before to this example yields the following:

Table 7.1: Example Prime Field Arithmetics for $p = 13$

Addition	Multiplication
$7 + 3 = 10 \pmod{13}$	$2 * 3 = 6 \pmod{13}$
$7 + 7 = 1 \pmod{13}$	$3 * 7 = 21 \pmod{13} = 8 \pmod{13}$
$-5 = 8 \pmod{13}$	$5^{-1} = 8 \pmod{13}$
	$5 * 8 = 1 \pmod{13}$
Subtraction	Division
$7 - 3 = 4 \pmod{13}$	$12/6 = 3 \pmod{13}$
$7 - 10 = 10 \pmod{13}$	$12/5 = 5 \pmod{13}$

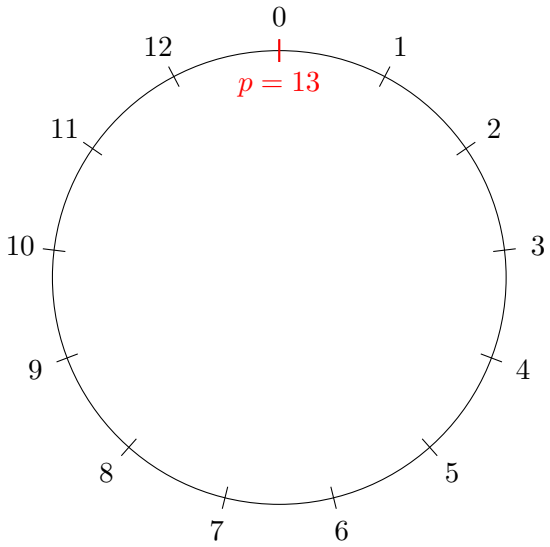


Figure 7.1: Prime Field Example with $p = 13$

Choice of Prime p: Recall that the prime field we work in when programming zk-SNARKs is defined as the field of discrete logarithms of a cyclic group of prime order. More precisely, the prime field modulus p is defined by the order of the cyclic group r. In practical zk-SNARK constructions, elliptic curves are commonly used as cyclic prime-order groups.

The Ethereum blockchain, for example, provides pre-compiled contracts for a Barreto-Naehrig curve (BN254) [27, 351], which enable cheap curve operation on the blockchain.[1] To be able to verify zk-SNARKs on Ethereum efficiently, we use this p as the modulus in ZoKrates programs. Operations inside a zk-SNARK program, constructed using that curve, will wrap around a prime p which equals the group order r of BN254, where:

$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$

This number has $bitwidth(p) = 254$. Hence, overflows can be expected to happen less frequently when compared to common data type bitwidths, such as 16 or 32 bits for unsigned integers `u16` or `u32`.

7.2 Provable Abstractions

As explained in Chapter 2, zk-SNARKs can theoretically be used to prove any **NP** statement. Concrete zk-SNARK protocols, however, rely on specific abstractions as characterizations of **NP**. Statements to be proven have to be expressed in these abstractions to avoid expensive reductions and maintain practicality.

[1] BN254 used to be referred to as BN128 where the postfix denoted the intended level of security of the curve. Since it does not achieve this level of security [202], the postfix now denotes the bitwidth of its field size.

In this section, we introduce two such abstractions that are closely related and commonly used in verifiable computation schemes: arithmetic circuits and R1CSs. To illustrate the connection between these circuits and programs, we additionally introduce straight-line programs (SLPs) as a third abstraction.

7.2.1 Arithmetic Circuits

In this section, we introduce arithmetic circuits as an abstraction commonly used in the context of verifiable computation protocols. We provide necessary background and define circuit satisfiability as the core property provable through zk-SNARKs.

Definition and Representation: A circuit is a directed, acyclic graph where nodes are called gates, and edges are called wires. Gates compute functions on their input wires to derive values for their output wires. Arithmetic circuits are circuits in which gates perform arithmetic operations, and wires have values that are generally not binary. Subsequently, we consider arithmetic circuits over the previously defined prime field $\mathbb{F}_p$ (see Section 7.1), i.e., $\mathbb{F}_p$-arithmetic circuits. Each wire carries a value from $\mathbb{F}_p$, and each gate performs a prime field operation, i.e., addition or multiplication.

Figure 7.2 shows an example for an arithmetic circuit with inputs $\vec{i} = i_0, \ldots, i_4$ and outputs $\vec{o} = o_0, \ldots, o_3$. From the inputs, intermediary wire values $\vec{w} = w_0, \ldots, w_5$ are calculated, which are then again used to derive outputs.

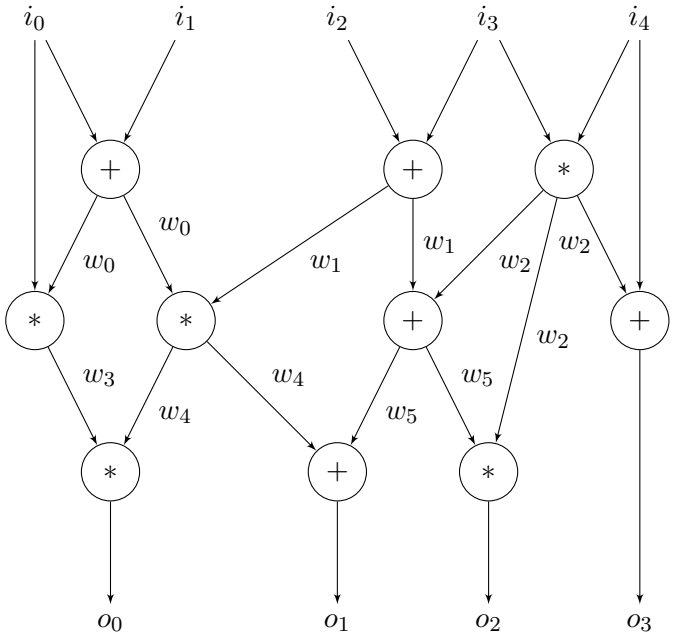


Figure 7.2: Example Arithmetic Circuit

We can represent this arithmetic circuit algebraically by defining the set of equations encoding

the gates:

$$\begin{aligned} w_0 &= i_0 + i_1 & w_1 &= i_2 + i_3 \\ w_2 &= i_3 * i_4 & w_3 &= i_0 * w_0 \\ w_4 &= w_0 * w_1 & w_5 &= w_1 + w_2 \\ o_0 &= w_3 * w_4 & o_1 &= w_4 + w_5 \\ o_2 &= w_5 * w_2 & o_3 &= w_2 + i_4 \end{aligned} \tag{7.1}$$

Note that scalar multiplication of one wire value can be modeled as repeated addition of the value with itself. Hence, it does not require any multiplication gates.

To describe a circuit concisely, we can represent it by the function it computes. We define $C : \mathbb{F}_p^n \rightarrow \mathbb{F}_p^m$ with inputs $i_0, \ldots, i_{n-1}$, and outputs $o_0, \ldots, o_{m-1}$. By solving C for $\vec{o}$, we observe that the circuit's outputs can be described through polynomials depending on the circuit inputs:

$$\begin{aligned} C_{o_0}(i_0, \cdots, i_4) &= o_0 = (i_0 * (i_0 + i_1)) * ((i_0 + i_1) * (i_2 + i_3)) \\ C_{o_1}(i_0, \cdots, i_4) &= o_1 = ((i_0 + i_1) * (i_2 + i_3)) + ((i_2 + i_3) + (i_3 * i_4)) \\ C_{o_2}(i_0, \cdots, i_4) &= o_2 = ((i_2 + i_3) + (i_3 * i_4)) * (i_3 * i_4) \\ C_{o_3}(i_0, \cdots, i_4) &= o_3 = (i_3 * i_4) + i_4 \end{aligned} \tag{7.2}$$

Arithmetic Circuit Satisfiability: A circuit is said to be satisfiable if there exists an assignment of wire variables complying with all the gates, their wiring, and the known values for inputs $\vec{i}$ and outputs $\vec{o}$. To provide an example, we formulate an instance of the satisfaction problem for the circuit in Figure 7.2: Given outputs and a (sub-)set of inputs, e.g., $\vec{o} = (2, 0, 2, 0)$ and $(i_0, i_1) = (1, 1)$, is there a set of auxiliary inputs and intermediate wire variables $\vec{w}$, so that the circuit is satisfied? The conditions defined in Equation (7.1) represent satisfiability of the example circuit algebraically: Any set of variables that satisfies the equations is a satisfying wire assignment.

Formally, we define the satisfiability problem for $\mathbb{F}_p$-arithmetic circuits through the following relation:

$$R_C = \{(\vec{i}, \vec{o}, \vec{w}) \in \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^h : C(\vec{i}, \vec{w}) = \vec{o}\,\}$$

Here, the values of intermediate variables $\vec{w}$ are referred to as witnesses or certificates, since they witness or certify that a variable assignment can be found so that the circuit is satisfied. Arithmetic circuit satisfiability is an NP-complete problem [17]: There is no efficient algorithm to solve the satisfaction problem, but a given solution $(\vec{i}, \vec{o}, \vec{w})$ can be verified efficiently, i.e., in polynomial time by a deterministic Turing machine.

7.2.2 Straight-line Programs

As previously established, zk-SNARK can be used to prove circuit satisfiability for given inputs $\vec{i}$ and outputs $\vec{o}$, i.e., to certify that there exists a witness $\vec{w}$ so that $C(\vec{i}, \vec{w}) = \vec{o}$ for a given arithmetic circuit C. In that process, the zero-knowledge property allows the witness to remain private to the prover.

We now describe how this result can be transferred from circuits to computations expressed as programs by describing the relationship between satisfied circuits and correct program execution.

Representing Circuits as Programs: The arithmetic circuit shown in Figure 7.2 not only defines satisfiability criteria as expressed in Equation (7.1), but also defines an order in which outputs can be transitively derived from inputs by computing intermediate results.

Respecting this order, we can explicitly represent this computation of outputs from inputs through a SLP. We follow the notation introduced by Savage [294], which we slightly adapt for our purposes. Such programs are called straight-line as they express programs without jumps and loops. The SLP notation resembles program representations in established programming languages.

Algorithm 1 shows a straight-line program which describes the example arithmetic circuit introduced in Figure 7.2.

A straight-line program consists of a list of numbered steps, where each step can be an input step, an output step, or a computation step. Input steps are written as ($l : READ\ i$), where l denotes the line number and input variable i. An output step which produces the result of line f is written as ($l : OUTPUT\ o$). Computation steps are denoted as ($l : OP\ f\ g$), where operator OP is applied to the results of steps f and g. As the arithmetic circuits that we represent through SLPs are defined over $\mathbb{F}_p$, the available operators are addition and multiplication, i.e., $OP \in \{*, +\}$. The operands for computation step l are the results of computation steps f, g. The condition $f, g < l$ always needs to hold to ensure that all required operands for a given operation have previously been computed.

Vice versa, the circuit can be obtained by graphing the straight-line program. This graph is always directed and acyclic since an SLP does not contain jumps, loops, or branches. In this mapping, each SLP computation step represents one arithmetic circuit gate, and thus, n-gate circuits correspond to programs with n computation steps. $READ$ and $OUTPUT$ operations define input and output wires.

Program Execution: A straight-line program with n inputs and m outputs computes a function $P : \mathbb{F}_p^n \to \mathbb{F}_p^m$. In the previous section, we demonstrated how to construct an SLP equivalent to a given arithmetic circuit, i.e., so that $P = C$. Thus, a circuit function C can be computed by executing the straight-line program step-by-step.

During the sequential execution of a SLP's steps, the evaluation of each line returns an intermediate value. We refer to this set of intermediate values as an *execution trace*. These execution trace

Algorithm 1 Equivalent straight-line program for example arithmetic circuit

Input: $\vec{i} = (i_0, \ldots, i_4)$
Output: $\vec{o} = (o_0, \ldots, o_3)$

1:	READ i_0	
2:	READ i_1	
3:	READ i_2	
4:	READ i_3	
5:	READ i_4	
6:	+ 1 2	$\triangleright\ w_0 = i_0 + i_1$
7:	+ 3 4	$\triangleright\ w_1 = i_2 + i_3$
8:	$*$ 4 5	$\triangleright\ w_2 = i_3 * i_4$
9:	$*$ 1 6	$\triangleright\ w_3 = i_0 * w_0$
10:	$*$ 6 7	$\triangleright\ w_4 = w_0 * w_1$
11:	+ 7 8	$\triangleright\ w_5 = w_1 + w_2$
12:	$*$ 9 10	$\triangleright\ o_0 = w_3 * w_4$
13:	+ 10 11	$\triangleright\ o_1 = w_4 + w_5$
14:	$*$ 11 8	$\triangleright\ o_2 = w_5 * w_2$
15:	+ 8 5	$\triangleright\ o_3 = w_2 + i_4$
16:	OUTPUT 12	
17:	OUTPUT 13	
18:	OUTPUT 14	
19:	OUTPUT 15	

variables coincide with the witness variables w in the arithmetic circuit representation.

This insight explains the following relation, which establishes a notion of computational correctness through circuit satisfiability:

Lemma 1: *An execution trace t for an SLP calculating P is valid if and only if $C(i, t) = P(i)$.*

We conclude that an execution trace retrieved by executing an SLP generates a witness that satisfies the corresponding arithmetic circuit in the process.

Proving Execution Correctness: Having reduced correct program execution to circuit satisfiability allows us to apply zk-SNARK protocols to computations. Instead of circuit satisfaction, we can now use a zk-SNARK to prove the validity of an SLP's execution trace: First, we execute an SLP step by step to obtain an execution trace for the SLP. Second, we transform this execution trace to a witness of the corresponding arithmetic circuit. Given this witness, we compute a zk-SNARK proof that certifies the satisfaction of this circuit. By Lemma 1, this is the case if and only if the execution trace was valid, i.e., the computation happened correctly.

Due to the zero-knowledge property of the employed computation scheme, proofs do not reveal execution traces. Only inputs and outputs of the SLP are revealed in the process.

The mental model of arithmetic circuits as an abstraction that represents computations explains the term *circuits* for provable programs, which is often encountered in literature on verifiable computations.

7.2.3 Rank-1 Constraint Systems

As the last abstraction, we introduce *rank-1 constraint systems (R1CSs)*. R1CS can be seen as a concise algebraic representation of arithmetic circuits. This abstraction is commonly used in implementations [54, 298] and cryptographic protocol specification [6, 35, 37]. Additionally, it illustrates the cost model for zk-SNARK computations, where only multiplications incur a cost, and additions are essentially free.

Definition and Representation: Equation (7.1) defines circuit satisfiability as a set of arithmetic constraints. Taking one of these equations as an example, we can rewrite it as follows into a product of variables:

$$\begin{aligned} & i_3 * i_4 = w_2 \\ \iff \quad & (1 * i_3) * (1 * i_4) = 1 * w_2 \end{aligned} \tag{7.3}$$

Equations of such structure can be conveniently expressed in vector notation, where $\langle \cdot, \cdot \rangle$ denotes the dot product. This results in the following representation for Equation (7.3):

$$\left\langle \begin{pmatrix} 0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}, \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \right\rangle * \left\langle \begin{pmatrix} 0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}, \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \right\rangle = \left\langle \begin{pmatrix} 0\\0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0 \end{pmatrix}, \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \right\rangle$$

We can generalize this idea by using matrix multiplication notation to express not only one but a set of constraints, i.e., a rank-1 constraint system. Applying this notation results in a concise arithmetic representation of the constraints given in Equation (7.1), where $\circ$ denotes the entry-wise or Hadamard product:

$$\begin{aligned} &\begin{pmatrix} 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \\ \circ &\begin{pmatrix} 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{pmatrix} \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \\ = &\begin{pmatrix} 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} 1\\i_0\\i_1\\i_2\\i_3\\i_4\\w_0\\w_1\\w_2\\w_3\\w_4\\w_5\\o_0\\o_1\\o_2\\o_3 \end{pmatrix} \end{aligned} \tag{7.4}$$

Rank-1 Constraint Satisfiability: Using this notation, we define the rank-1 constraint satisfiability problem as follows:

Can a vector of inputs and outputs $(\vec{i}, \vec{o}) \in \mathbb{F}_p^{n+m}$ be complemented by a witness vector $\vec{w} \in \mathbb{F}_p^h$, so that

$$\mathbf{A}(1, \vec{i}, \vec{o}, \vec{w})^\intercal \circ \mathbf{B}(1, \vec{i}, \vec{o}, \vec{w})^\intercal = \mathbf{C}(1, \vec{i}, \vec{o}, \vec{w})^\intercal \tag{7.5}$$

where $\circ$ denotes the entry-wise or Hadamard product and $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are matrices over $\mathbb{F}_p$. 1 is explicitly encoded in the vector to support multiplication with and addition of scalar values within constraints.

Constraint System Optimization: We now demonstrate how addition constraints can be eliminated from R1CS through elegant encoding without changing satisfiability of the system. This optimization directly benefits the efficiency of zk-SNARK proofs for R1CS.

As we defined R1CS over $\mathbb{F}_p$, there exist addition and multiplication constraints. The following example shows how two constraints from example Equation (7.4), one addition and one multiplication, can be transformed so that the addition condition is included in the multiplication constraint. The addition constraint is eliminated through leveraging the R1CS' structure resulting in a constraint system of only one constraint:

$$\begin{aligned}
& i_0 + i_1 = w_0 \\
& i_0 * w_0 = w_3 \\
\iff \quad & i_0 * (i_0 + i_1) = w_3 \\
\iff \quad & (1 * i_0) * (1 * i_0 + 1 * i_1) = (1 * w_3) \\
\iff \quad & (0,1,0,0) \begin{pmatrix} 1 \\ i_0 \\ i_1 \\ w_3 \end{pmatrix} * (0,1,1,0) \begin{pmatrix} 1 \\ i_0 \\ i_1 \\ w_3 \end{pmatrix} = (0,0,0,1) \begin{pmatrix} 1 \\ i_0 \\ i_1 \\ w_3 \end{pmatrix}
\end{aligned}$$

Here, instead of only multiplying two variables, the multiplication constraint encodes the product of two linear combinations of variables.

More generally, we observe that addition constraints represent linear combinations of variables. Linear combinations can be expressed through a linear function $f(x) = \mathbf{M}x^\intercal$, with $f : F^n \to F^m$ for an $m \times n$ matrix $\mathbf{M}$ over $\mathbb{F}_p$. This is precisely the structure on both sides of the Hadamard product $\circ$ in the R1CS definition (see Equation (7.5)). We can leverage the R1CS matrices $\mathbf{A}$,$\mathbf{B}$, and $\mathbf{C}$ to encode conditions in the form of linear combinations within multiplication constraints, thereby eliminating any explicit addition constraints.

Applying this idea, i.e., encoding additions within multiplication constraints, to the example provided in Equation (7.4), we arrive at an optimized R1CS version without addition constraints:

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix} 1 \\ i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ w_2 \\ w_3 \\ w_4 \\ o_0 \\ o_1 \\ o_2 \\ o_3 \end{pmatrix}
$$

$$
\circ
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix} 1 \\ i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ w_2 \\ w_3 \\ w_4 \\ o_0 \\ o_1 \\ o_2 \\ o_3 \end{pmatrix}
$$

$$
=
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix} 1 \\ i_0 \\ i_1 \\ i_2 \\ i_3 \\ i_4 \\ w_2 \\ w_3 \\ w_4 \\ o_0 \\ o_1 \\ o_2 \\ o_3 \end{pmatrix}
$$

Analogously to the case of arithmetic circuits, the correctness of program execution can be reduced to R1CS satisfiability. R1CS satisfaction can then directly be proved through zk-SNARKs. The time and space complexity of the setup, witness computation, and proof generation steps directly depends on the number of constraints in the system. Therefore, the optimization introduced in this section, which reduces the number of constraints in an R1CS without changing its satisfiability, is crucial for efficiency.

An optimized constraint system only contains multiplication gates. Hence, only these constraints contribute to the cost associated with proving satisfiability through zk-SNARKs. Due to the equivalence of R1CS and arithmetic circuit satisfiability, we can directly transfer this result: In an arithmetic circuit, only multiplication gates contribute to the complexity of zk-SNARK proofs of satisfiability. Addition gates do not need to be considered. Refer to [268] for an alternative derivation of this result.

Due to their brevity and conciseness, R1CSs have evolved as a de-facto standard input format for zk-SNARK implementations. As a result of their notation, which uses standard linear algebra, they also lend themselves well to cryptographic protocol descriptions and theoretical analysis.

7.3 Expressiveness and Efficiency

In this section, we examine which computations can be proved with zk-SNARKs from a theoretical, as well as a practical perspective.

We show how circuit-external witness derivation can be used to significantly reduce the number of multiplication-gates, thereby ensuring practicality with regards to proving.

We use the arithmetic circuit abstraction in this analysis (see Section 7.2.1). In the previous sections, we established the equivalence of arithmetic circuit satisfaction, R1CS satisfaction, and the correctness of an SLP's execution trace. Hence, the insights gained in this section universally apply for all other abstractions as well.

7.3.1 Provable Computations

In this section, we show that the correct computation of any finite function can be proved through zk-SNARKs. Furthermore, we explain why our intuitive notion of computability coincides with polynomial-sized circuits.

Computable Functions: As established in Section 7.2.1, an arithmetic circuit computes a function $f_C : \mathbb{F}_p^n \to \mathbb{F}_p^m$ and zk-SNARKs can prove arithmetic circuit satisfaction. To answer the question of which computations can be proved through zkSNARK, we thus need to determine which functions can be expressed as a circuit satisfaction problem.

In a first step, we show that arithmetic circuits can compute any finite function: Analogously to our definition of arithmetic circuits in Section 7.2.1, we define a boolean circuit C as circuit with wire values from $\{0, 1\}$ and gates that compute boolean operators $\{\wedge, \vee, \neg\}$, i.e., the boolean functions AND, OR and NOT. A boolean circuit with n input and m output wires computes a function $f_C : \{0, 1\}^n \to \{0, 1\}^m$.

A finite function is a function with a fixed number of inputs and outputs. Boolean circuits can compute every finite function. More precisely, for a given finite function $f : \{0, 1\}^n \to \{0, 1\}^m$, there exists a boolean circuit that computes f with at most $m * n2^n$ gates [25]. Intuitively, this standard result can be understood by constructing the truth table for f and describing the result columns as logical formulas in *disjunctive normal form (DNF)* [294].

Having established that every finite function can be computed by a boolean circuit, we now demonstrate that this result also holds for arithmetic circuits. We provide a constructive method for the efficient simulation of boolean circuits in arithmetic circuits, i.e., with polynomial overhead in circuit size. Refer to [147, 192, 309, 310] for surveys of arithmetic and boolean circuit complexity.

A set of boolean functions is *functionally complete* if every boolean function can be expressed using only functions from this set [17]. Subsequently, we use the boolean functions NOT, AND, and OR as a functionally complete set, as they coincide with our boolean circuit definition. The same results can be obtained using any functionally complete set, e.g., the set only containing the $NAND$ function.

We can emulate the boolean NOT, AND, and OR functions in $\mathbb{F}_p$-arithmetics as follows:

$$\begin{aligned} NOT(a) &= 1 - a \\ AND(a, b) &= a * b \\ OR(a, b) &= 1 - (1 - a) * (1 - b) \end{aligned} \tag{7.6}$$

Table 7.2 visualizes this relationship by providing truth tables for the boolean and arithmetic functions.

Table 7.2: Truth Tables for Functionally Complete Set of Boolean Operators

a	NOT $\overbrace{\neg a}$	$(1-a)$
1	0	0
0	1	1

a	b	AND $\overbrace{a \wedge b}$	$(a * b)$
1	1	1	1
1	0	0	0
0	1	0	0
0	0	0	0

a	b	OR $\overbrace{a \vee b}$	$1-(1-a)*(1-b)$
1	1	1	1
1	0	1	1
0	1	1	1
0	0	0	0

Both, the NOT and AND $\mathbb{F}_p$-arithmetic functions in Equation (7.6) contain exactly one operation. Thus, exactly one arithmetic gate is required for the simulation of these boolean gates within an arithmetic circuit. By the same logic, the simulation of the OR function needs three boolean gates. Hence, the emulation is efficient with regard to circuit size. □

The simulation is also possible in the reverse direction: Boolean circuits can efficiently emulate arithmetic circuits over prime fields. Here, the transformation uses the textbook combinatorial circuits for addition, multiplication and modular division. As this result is purely supplementary in our context, we do not provide an in-depth description. For an analysis of the complexity of this simulation with regards to circuit size, refer to [309].

In summary, boolean circuits can efficiently emulate arithmetic circuits over prime fields and vice versa. We can compute any finite function in arithmetic circuits and prove that the computation happened correctly using zk-SNARKs.

Programming Languages and Turing Machines: In this section, we describe the relationship between programming languages, Turing machines, and circuits, as well as the resulting implications. This discussion is based on complexity-theoretic standard results, we refer to literature for formal definitions and proofs [17, 25, 294].

A computer program P consists of a collection of instructions and P computes a function f which maps inputs $\vec{i}$ to outputs $\vec{o}$. Without loss of generality, we define input and outputs to be binary, i.e., $i_i, o_i \in \{0, 1\}$. This condition can always be guaranteed through a suitable encoding [17].

Established programming languages, e.g., Java and C, are Turing-equivalent, i.e., programs specified in these languages compute functions that can be computed by Turing machines and vice

versa. The Turing machine model of computation is uniform: A program computes a function f for arbitrary input sizes. Formally, we write $f : \{0,1\}^* \to \{0,1\}^*$. Since the inputs and outputs are not bounded to any length, this function is *infinite*.

Previously, we showed how any finite function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by an arithmetic circuit of exponential size. However, this circuit is specific to one input size n, and computing f for different input sizes requires different circuits: The circuit model of computation is non-uniform.

According to the Church-Turing thesis, the functions computable by Turing machines coincide with the human notion of effectively calculable functions. Any algorithm can be simulated by a Turing machine. Hence, the relation between this model of computation and the non-uniform circuit model is insightful with regards to the expressiveness of the abstractions.

In section Section 7.2.2, we introduced straight-line programs (SLPs) as a circuit-equivalent model of computation, which consists of a simple list of instructions. Turing machines generalize this model: They achieve uniformity by being able to execute programs containing loops and arrays. Loops allow the same instruction to be executed multiple times, and arrays can store an arbitrarily long sequence of variables.

By fixing the input size of a program in a Turing-equivalent language to n, we make it a finite problem again, which allows unrolling the program to a circuit: For a program which computes f in $t(n)$ steps, there exists a circuit of size $poly(t(n))$ that computes $f\big|_n$, i.e., a restriction of f to input size n [25]. The intuition is as follows: Using a standard transformation [17], the original program can be rewritten efficiently to an *oblivious* program, so that fixing the input size n uniquely determines the number of iterations for all loops. Then, the loop can be unrolled into a straight-line program by copying instructions within loops in the original program and then appending them to the SLP once for every iteration.

We can further generalize this relationship: An infinite function f can be calculated by a *circuit family*, an infinite sequence of different circuits $\{C_n : n \in \mathbb{N}\} = (C_0, C_1, \ldots)$, where C_n computes $f_n : \{0,1\}^n \to \{0,1\}^m$. If f is computable by a (deterministic) Turing machine in polynomial time $t(n)$, f can be computed by a circuit family consisting of polynomially-sized circuits ($size(C_n) = poly(t(n))$), where $C_n = f\big|_n$ [25].

In summary, computations we can prove with zk-SNARKs have a fixed and finite number of inputs. Given a program in any Turing-equivalent programming language (without I/O), we can prove the correct execution of that program for a fixed input size n by unrolling it into a circuit and proving satisfiability. However, we have to expect that the resulting circuits are of exponential size and hence impractical to work with. If a program to be unrolled is in the complexity class $\mathbf{P}$, the resulting circuit is of size polynomial in the number of inputs n.

7.3.2 Proving Computations Efficiently

In the previous section, we obtained results on the power and expressiveness of arithmetic circuits and equivalent abstractions. We mainly derived existential statements and analyzed asymptotic

complexity. From our analysis, we concluded that for any finite function f, we can construct an arithmetic circuit that computes this function.

In the context of real-world zk-SNARK applications, however, asymptotic results are less relevant, and concrete efficiency is crucial to ensure practicality. Thus, in this section, we discuss approaches to enable fast execution and proving of computations.

At the start of this section, we relate the notions of executing and proving computations to the equivalent problems in the previously introduced abstractions and discuss complexity as well as efficiency. We observe that deriving a solution for a given circuit and proving that solution's correctness are orthogonal problems.

Based on this insight, we provide two important results that enable more efficient circuits:

- Circuit-external Witness Derivation
- Boolean Circuit Embedding

Executing and Proving: In the context of this work, zk-SNARKs are used to prove the correctness of a computation. A proof attests correctness of a straight-line program's execution trace, or equivalently, the satisfaction of an arithmetic circuit for a wire-value assignment. Before such a proof can be generated, a satisfying wire-value assignment has to be found, i.e., the computation needs to be executed. Hence, the overall efficiency is determined by two steps, a) program execution and b) proof generation.

In this section, we discuss the complexity of both steps and the implied impact on overall efficiency. We explain the relationship between satisfiability and solving an arithmetic circuit by discussing two related, but distinct problems, arithmetic circuit satisfiability (ACSAT) and the circuit value problem (CVP).

a) **Program Execution:** We generalize our example for the arithmetic circuit satisfiability problem from Section 7.2.1: For an arithmetic circuit instance $C : \mathbb{F}_p^n \times \mathbb{F}_p^h \rightarrow \mathbb{F}_p^m$, and outputs $\vec{o}$, $|\vec{o}| = m$, the arithmetic circuit satisfiability problem

$$ACSAT(C, \vec{o}) = 1$$

is defined by its set of satisfying inputs

$$L_C = \{\vec{i} \in \mathbb{F}_p^n : \exists \vec{w} \in \mathbb{F}_p^h \text{ s.t. } C(\vec{i}, \vec{w}) = \vec{o}\ \}$$

or equivalently by the relation

$$R_C = \{(\vec{i}, \vec{w}) \in \mathbb{F}_p^n \times \mathbb{F}_p^h : C(\vec{i}, \vec{w}) = \vec{o}\ \}$$

with inputs $\vec{i}$, $|\vec{i}| = n$ and witness $\vec{w} \in \mathbb{F}_p^h$. $ACSAT(C, \vec{o})$ is 1 if and only if the circuit C is satisfiable for $\vec{o}$ and 0 otherwise.

zk-SNARKs allow us to prove satisfiability for a problem instance. More precisely, we can attest that there exists a variable assignment $(\vec{i}, \vec{o}, \vec{w})$ for the wires in the circuit so that all gates are satisfied, i.e., $C(\vec{i}, \vec{w}) = \vec{o}$.

Up to this point, we have not discussed how a variable assignment satisfying this condition can be found. Given an arbitrary circuit instance C and outputs $\vec{o}$, $ACSAT(C, \vec{o})$[2] is $\mathbf{NP}$-complete [151]. Thus, there is no efficient algorithm for finding a satisfying variable assignment (unless $\mathbf{P} = \mathbf{NP}$). The best strategy is to guess a vector of input values $\vec{i}$ and to check whether it satisfies the circuit, which has complexity $O(2^n)$.

As previously explained, executing a computation is equivalent to finding a satisfying solution for an arithmetic circuit. At first sight, the $\mathbf{NP}$-completeness of $ACSAT$ gives little hope for solving this problem efficiently. However, when executing a computation, the inputs are always known, which significantly simplifies the problem at hand:

Deciding whether input $\vec{i}$ satisfies $ACSAT(C, \vec{o})$ is called the *circuit value problem (CVP)*. We denote $CVP(C, \vec{i}, \vec{o}) = 1 \iff \exists \vec{w} \in \mathbb{F}_p^h$ s.t. $C(\vec{i}, \vec{w}) = \vec{o}$. It is intuitively clear that $CVP \in \mathbb{P}$ since every gate can be evaluated in constant time, and the inputs to each gate transitively depend on the circuit's inputs. We refer to [294] for a formal proof. A solution $\vec{i}$ has a certificate $\vec{w}$, which allows a polynomial-time check by a non-deterministic Turing machine. More precisely, CVP's complexity is linear in the circuit's size s, i.e., $O(s)$.

b) **Proof Generation:** As described in Chapter 2, the complexity of proving satisfaction of an arithmetic circuit through a state-of-the-art zk-SNARK is $O(n \log n)$ (e.g., for the Groth16 scheme [164]), where n is the number of multiplication gates. Thus, the concrete efficiency of this proof generation step is directly determined by n. We say that a circuit C is more efficient than C' if and only if both compute the same function and $n_C < n'_C$.

Note that, unlike in program execution, the complexity depends on the number of multiplication gates instead of circuit size. This result implies different objective functions depending on whether we optimize for proving or execution speed. However, as proving times dominate execution times in practice, our optimization techniques always minimize the number of multiplication gates. Refer to Chapter 11 for empirical evidence.

Circuit-external Witness Derivation: We now introduce *circuit-external witness derivation* as a fundamental approach to designing efficient circuits by reducing the number of multiplication gates. Here, we leverage the independence of program execution (deriving a witness) and proof generation.

To study this idea, we revisit the basic arithmetic operations defined for $\mathbb{F}_p$ and how they can be represented in arithmetic circuits. As introduced in Section 7.1, these operations are addition, multiplication, subtraction, and division. Our definitions of arithmetic circuits, and R1CS respectively, natively only support addition and multiplication, however. This observation raises the question of how subtraction and division can be represented.

One possibility, as previously explained, is to simulate binary subtraction and division circuits within an arithmetic circuit. However, this leads to a significant increase in circuit size for a very

[2] Oftentimes, without loss of generality, circuit instances C in satisfiability problem formulations are assumed to have only one boolean output that is `true` for satisfying variable assignments. Thus, $\vec{o}$ is not part of the problem's definition in these cases.

basic operation, and practicality deteriorates. This approach is commonly applied in the context of secure multiparty computations, where circuits are directly evaluated to compute a function (see Chapter 2). In the context of zk-SNARKs, evaluating a circuit, i.e., finding a witness and proving witness validity are entirely independent. We can leverage this domain-specific property to avoid complex boolean simulations, as subsequently demonstrated.

Consider the following example R1CS constraint:

$$i_0/i_1 = o_0$$

As noted before, we can not explicitly perform divisions in a R1CS, or an arithmetic circuit. Applying an equivalent transformation, we can formulate the relation between the variables through a multiplication by demanding:

$$o_0 * i_1 = i_0 \tag{7.7}$$

This constraint ensures correctness[3]; the same variable values satisfy the constraint system. At the same time, this transformation changes the circuit's structure:

The specification of the circuit is no longer a *constructive*: The values i_0, i_1 are known and o_0 needs to be found, but Equation (7.7) calculates i_0 from inputs i_1, o_0. Solving the circuit value problem for this circuit is no longer in $\mathbf{P}$; the variable value for o_0 would need to be guessed, as it is not computable from inputs within the circuit.

Fortunately, we can solve this issue by leveraging the fact that proving only requires a satisfying variable assignment and this assignment does not have to be computed within a circuit: We compute $o_0 = i_0/i_1$ outside of the circuit and provide the result as *advice* or *auxiliary inputs* [35] during evaluation. Since the transformed constraint provided in Equation (7.7) remains unaltered, relying on external computations for witness derivation never impairs correctness.

We observe that such an evaluation problem can be interpreted as solving CVP for an incomplete set of inputs $\vec{i}_{inc}$. Since the missing auxiliary inputs $\vec{i}_{aux}$ need to be guessed, the CVP problem deteriorates to a $\mathbf{NP}$-complete satisfiability problem $CVP(C, \vec{i}_{inc}, \vec{o})$, which can be considered a variant of $ACSAT(C, \vec{i}_{inc}, \vec{o})$. To restore efficient witness derivability, we define a *solver function* $f\colon \mathbb{F}_p^n \to \mathbb{F}_p^k$ which maps inputs to auxiliary inputs $f(\vec{i}_{inc}) = \vec{i}_{aux}$. Using this solver function, we can compute the missing inputs outside the circuit and obtain an efficiently solvable problem again: $CVP(C, (\vec{i}_{inc}, f(\vec{i}_{inc})), \vec{o}) = CVP(C, (\vec{i}_{inc}, \vec{i}_{aux}), \vec{o}) \in \mathbf{P}$.

The same approach could be applied to efficiently model subtraction, i.e., finding the additive inverse. Consider the following example:

$$\begin{aligned} & i_0 - i_1 = o_0 \\ \iff\ & o_0 + i_1 = i_0 \end{aligned}$$

[3] For simplicity of the argument, we neglect that this transformation introduces $i_1 = 0$ as a trivial solution. This solution can be excluded by adding the constraint $i_0 * w = 1$, since $i_1 \neq 0 \iff \exists w : i_1 * w = 1$.

However, there is an even more efficient solution. Since additive inversion is equivalent to scalar multiplication by -1:

$$\begin{aligned} i_0 - i_1 &= o_0 \\ i_0 + (-i_1) &= o_0 \\ \iff i_0 + (-1) * i_1 &= o_0 \end{aligned}$$

As demonstrated, we can apply an equivalent transformation to the constraint system so that it only uses addition without changing the subtracted variable at all. Division, in contrast, requires multiplicative inversion, which cannot be translated to a modification of scalars and thus benefits from external advice.

Boolean Circuit Embedding: Functions consisting of additions and multiplications of field elements can be computed naturally and very efficiently in arithmetic circuits. This includes integer arithmetics and cryptographic operations defined on finite fields, e.g., elliptic curve operations (see Section 8.3.2). Simulating an equivalent boolean circuit would cause considerable overhead in circuit size.

Some operations, however, have no straight-forward arithmetic representation: An example of this is the problem of deciding whether a field element x is even or odd. There is no obvious way to compute an answer using only a set of addition and multiplication gates.

In a boolean circuits with wires containing a bit representation of x, in contrast, a simple check of the wire representing the least-significant bit answers the question: Let the values $x_{n-1}, \ldots, x_0 \in \{0,1\}$ be a binary decomposition of $x \in \mathbb{F}_p$, i.e., $x = \sum_{i=0}^{n-1} x_i * 2^i$. We can then simply define the following function:

$$even(x) = \begin{cases} true & \text{if } x_0 = 0 \\ false & \text{else} \end{cases}$$

We conclude that, depending on the problem at hand, a solution using only $\mathbb{F}_p$-arithmetics may be more efficient, but not always possible in arithmetic circuits. Up to now, this would leave us with only one option, that is simulating a boolean circuit through an arithmetic circuit and thus to only operate on variables with values $\{0,1\} \subset \mathbb{F}_p$, which does not use the abstraction's power to its fullest extent. To improve on this result, we now show how a boolean circuit can be embedded in an arithmetic circuit, i.e., how we can mix both abstractions. Mixing allows us to compute both arithmetic and boolean functions in one arithmetic circuit. This idea has initially been introduced by Parno et al. as a *split gate* [268]. For simplicity of the argument, we use the R1CS abstraction, which is equivalent to the arithmetic circuit model.

We introduce the concept of a *binary decomposition*, which consists of a set of constraints of two types:

- *Decomposition Constraint:* Let $x \in \mathbb{F}_p$. We assert that a set of variables $x_{n-1}, \ldots, x_0 \in \mathbb{F}_p$ is a decomposition of x by introducting the following constraint:

$$x = \sum_{i=0}^{n-1} x_i * 2^i \tag{7.8}$$

  where, $n = bitwidth(|\mathbb{F}_p|) = \lceil log_2(p) \rceil$. Only one constraint is required to assert this equality as the right side of Equation (7.8) represents a linear combination.

- *Binarity Constraints:* To ensure the decomposition is actually a binary decomposition, we need to restrict the field element's values to $\{0, 1\} \subset \mathbb{F}_p$. We can assert that through the following set of constraints:

$$(1 - x_i) * x_i == 0, \quad \forall i \in \{0, \ldots, n-1\}$$

  Hence, binarity is ensured by n constraints.

Together, this binary decomposition requires $n + 1$ constraints; or $n + 1$ multiplication gates in the equivalent arithmetic circuit model.

To obtain a witness for the binary decomposition constraints, we can use the circuit-external witness derivation approach previously introduced: We calculate values $x_0, \ldots, x_{n-1}$ outside the circuit, which is a trivial decimal-to-binary conversion problem, and provide them as auxiliary inputs.

To be able to use the binary results of an embedded boolean circuit within $\mathbb{F}_p$-arithmetics again, the binary variables $x_{k-1}, \ldots, x_0 \in \{0, 1\}$ need to be recombined to a field element x. This *binary recomposition* consists of only one constraint and is defined analogously to Equation (7.8):

$$\sum_{i=0}^{k-1} x_i * 2^i = x$$

Intuitively, one may assume that it is sufficient to require $k \leq n$ to ensure that the field element has a unique representation in $\mathbb{F}_p$. Yet, this is only true for $k < n$: The bit decomposition $n = bitwidth(|\mathbb{F}_p|) = \lceil log_2(p) \rceil$ allows a binary decomposition to represent numbers $\in [0; 2^n - 1]$. This interval contains numbers $[p; 2^n - 1)$, which, according to the rules of modular $\mathbb{F}_p$-arithmetics are mapped to $[0; 2^n - 1 - p]$. For this interval, binary decompositions $b_{n-1}, \ldots, b_0$ are not unique. Thus, an additional in-circuit bit-level check is required that asserts $x < p$ to ensure that only a decomposition that is also valid in $\mathbb{N}$ is accepted, which restores uniqueness. If we assume $k < n$, the decomposition has less bits $b_{k-1}, \ldots, b_0$ than required to represent p. Thus this decomposition only represents numbers $\in [0; 2^k - 1]$ and $2^k - 1 <= 2^{n-1} - 1 < p$. Hence, the decomposition is unique.

7.4 The ZoKrates Intermediate Representation

In the previous section, we explained how we can leverage circuit-external information to reduce the number of multiplication gates in a circuit, thereby reducing the cost of zk-SNARK proof

generation. This input-dependent information is provided as part of the witness and not derived within the circuit. This approach — at the same time — implies that a satisfying variable assignment can not be found by directly evaluating an arithmetic circuit. Additional calculations based on the input variables are required.

This peculiarity is not reflected in any of the abstractions introduced in Sections 7.2.1 to 7.2.3, which encode satisfiability, but can not be evaluated efficiently for circuits that rely on circuit-external witness derivation. However, this witness derivability property is crucial in practice: Being able to prove that a variable assignment satisfies a circuit is barely useful without a way to derive said assignment efficiently. For a given circuit, there needs to be an efficient way for evaluation, or equivalently, the encoded program needs to be executable efficiently.

To address this limitation of existing low-level abstractions, we introduce the *ZoKrates intermediate representation (ZIR)*, a new abstraction that extends R1CS to support both, efficient program execution and proof generation. It encodes a rank-1 constraint satisfaction problem and additional directives, which represent calls to solver functions, to support efficient circuit-external witness derivation (see Section 7.3.2).

We specify the ZIR as a structured format that serves as the compilation target for the ZoKrates Language (see Chapter 8). Programs compiled to ZIR can be efficiently run by the ZoKrates Interpreter, and the resulting execution trace can be proved correct using a ZoKrates backend. For a detailed description, refer to Chapter 9.

The ZIR has two types of statements, *constraints* and *directives*. Constraints have a dual function:

- They specify correctness conditions and are transformed into an R1CS by the ZoKrates compiler and subsequently used in the proving process.
- They are used during evaluation by the ZoKrates Interpreter to derive a satisfying variable assignment for the R1CS generated in the compilation step. However, this is not sufficient in constraint systems relying on circuit-external witness derivation.

Directives tell the interpreter to call an external solver function to obtain variable values that could not be derived by constraint evaluation due to the reliance on circuit-external witness derivation.

The ZIR Format: Using this definition and the basic ZIR ideas previously introduced, we now provide a more formal specification of the ZIR format. The ZIR format is a binary format but can be represented as human-readable text as described hereafter.

The ZIR format starts with a program signature that contains definition of input variables and the number of outputs in the first line. The last line defines output variables. Enclosed by these

definitions is a list of statements, resulting in the following structure:

```
def main([variables]) -> (#outputs):
[statements]
return [variables]
```

Each statement is either a *constraint*, or a *directive*.

Constraint: A constraint can be understood as an assertion that needs to be fulfilled at all times. Linear combinations are a core building block of constraints in the ZIR. We define a linear combination analogously to the R1CS case:

$$lincomb := \langle \vec{a}, \vec{x} \rangle = \sum_{i=0}^{n-1} a_i * x_i$$

with scalars $\vec{a} \in \mathbb{F}_p^n$ and variables $\vec{x} \in \mathbb{F}_p^n$, where $n = |\vec{a}| = |\vec{x}|$.

There are two types of constraints:

- *Definition*: A definition is of the structure

  ```
  variable == lincomb * lincomb
  ```

  where a variable is computed as a product of linear combinations. This is evaluated differently, depending on context: a) in proof generation, a definition is translated to a R1CS constraint $variable = lincomb * lincomb$, and b) in program execution, the interpreter simply calculates the value of `variable` by evaluating the right side of the equation. Intuitively, statements represent forward-computable gates in a circuit.

- *Assertion*: An assertion is of the structure

  ```
  lincomb == lincomb * lincomb
  ```

  i.e., the left side is a linear combination instead of a variable. In proof generation this is translated to an R1CS constraint $lincomb = lincomb * lincomb$; assertions do not have to be and are not interpreted.

Directive: Directives are of the form:

```
# [variables] = solver_function([variables])
```

A directive is only evaluated during interpretation and defines a function call to a solver function to be called with known inputs to derive witness information. The results of this call are assigned to the associated witness variables.

Example: We conclude this section with an example of a ZIR. Recall the sample arithmetic circuit from Figure 7.2 and its constraint representation provided in Equation (7.1). In Listing 7.1, we provide a ZoKrates implementation of this circuit, where each gate is evaluated in an arithmetic expression.

Listing 7.1: Example Arithmetic Circuit in ZoKrates

```
def main(field i_0, field i_1, field i_2, field i_3, field i_4) -> (field,
    field, field, field):

  field w_0 = i_0 + i_1
  field w_1 = i_2 + i_3
  field w_2 = i_3 * i_4
  field w_3 = i_0 * w_0
  field w_4 = w_0 * w_1
  field w_5 = w_1 + w_2

return w_3 * w_4, w_4 + w_5,  w_5 * w_2, w_2 + i_4
```

Compiling this example with ZoKrates yields the ZoKrates Internal Representation displayed in Listing 7.2. It contains seven constraints, which is exactly the number we also obtained by hand-optimizing the R1CS example in Section 7.2.3. As the arithmetic circuit implementation only uses addition and multiplication operations, which do not rely on external witness-derivation, the ZIR does not contain any directives.

Listing 7.2: ZoKrates Internal Representation compiled from Example Arithmetic Circuit

```
def main(_0, _1, _2, _3, _4) -> (4):
  (1 * _3) * (1 * _4) == 1 * _7
  (1 * _0) * (1 * _0 + 1 * _1) == 1 * _8
  (1 * _0 + 1 * _1) * (1 * _2 + 1 * _3) == 1 * _9
  (1 * _8) * (1 * _9) == 1 * ~out_0
  (1 * ~one) * (1 * _2 + 1 * _3 + 1 * _7 + 1 * _9) == 1 * ~out_1
  (1 * _2 + 1 * _3 + 1 * _7) * (1 * _7) == 1 * ~out_2
  (1 * ~one) * (1 * _4 + 1 * _7) == 1 * ~out_3
   return ~out_0, ~out_1, ~out_2, ~out_3
```

Obviously, this example is rather trivial, and meaningful programs require more than simple arithmetics. Therefore, the ZoKrates language provides additional types and abstractions developers are familiar with, e.g., structs, loops, conditions. These features will be covered in detail in Chapter 8. Nevertheless, the ZIR serves as the common abstraction all ZoKrates programs are compiled to.

7.5 Conclusion

In this chapter, we introduced arithmetic circuits, straight-line programs, and rank-1 constraint systems over prime fields as fundamental abstractions that allow the formulation of $\mathbf{NP}$-statements that can be proved with zk-SNARKs. We explained how these abstractions can be optimized by

avoiding multiplications of variables whenever possible.

A complexity-theoretic analysis showed that any finite function can be computed and proved with these abstractions. However, the space-complexity of this computation may be exponential and thus inefficient to prove in practice. To address this problem, we proposed *circuit-external witness derivation* and *boolean circuit embedding* as approaches to reduce space complexity and thereby improve prover efficiency.

Our analysis of the existing abstractions led to the insight that they insufficiently cater to the need for efficient execution of programs leveraging the previously introduced optimization approaches. To address this limitation, we introduced the ZoKrates intermediate representation (ZIR) as a low-level representation which combines provability and efficient solving.

In the subsequent chapter, building on the results obtained in this chapter, we introduce the ZoKrates language.

CHAPTER 8

The ZoKrates Language

As explained in the previous chapter, the low-level abstractions used in the specification of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) verifiable computation schemes are hard to program and insufficiently support external witness derivation. The ZoKrates intermediate representation (ZIR) solves this second issue but does not simplify the specification of computations.

In this section, we introduce the domain-specific ZoKrates language, a high-level programming abstraction that is designed to translate into efficiently provable programs represented in the ZIR format. Hiding away the intricacies of specifying computations in low-level abstractions, our language enables blockchain developers to program off-chain computations in a way that is more convenient, less error-prone, and closer to what they are used to. We start by discussing design principles and goals before defining the language on a syntactic and semantic level. After introducing the ZoKrates Standard Library, we examine the compilation process from the ZoKrates language to the ZoKrates intermediate representation in detail.

Although this language is a core part of the ZoKrates framework, the other elements of the toolchain described in Chapters 6 and 9 are also compatible with other ways of circuit specification (see Section 3.2.3). Hence, the ZoKrates language can be considered an important building block, but not a strict necessity in order to perform off-chain computations with ZoKrates.

8.1 Challenges and Design Principles

Established general-purpose programming languages, such as C [196], C++ [186], Java [16], JavaScript [114] and many more, are Turing-equivalent, i.e., they can express any computation executable by a universal Turing machine and a universal Turing machine can compute any computation expressed in the language. A program expressed in such languages can handle arbitrary numbers of inputs – it is uniform. The ZoKrates intermediate representation, in contrast, is not Turing-equivalent; it does not support conditional jumps or unbounded loops. A computation is

essentially represented by a long list of statements and has a defined number of inputs. Refer to Section 7.3 for an in-depth discussion of this relation.

Due to these fundamental differences, a language that translates into the ZoKrates intermediate representation cannot be Turing-equivalent. For a program in a language more expressive than ZIR, additional constraints would need to be provided to restore parity, e.g., upper bounds for loops would need to be specified, so that the program could be unrolled into a straight-line program. Nevertheless, this insight does not prevent a high-level language from exposing nice and familiar syntax to developers: Some syntactic sugar that does not expand expressiveness, e.g., bounded loops or conditionals, can be supported, as we will see in later sections of this chapter.

Besides this fundamental mismatch with regards to expressiveness, another fundamental difference motivates the need for a high-level abstraction, i.e., a domain-specific language. Aforementioned general-purpose programming languages target modern instruction set architectures (ISAs) implemented in digital microprocessors. They directly compile to such instructions or are executed by interpreters, which themselves leverage the ISA. This common target abstraction influences the languages, e.g., their type systems and operator sets: Data types are based on binary representations and the available operators are generally efficiently computable. In the ZoKrates intermediate representation, in contrast, prime field elements are the native type, and binary representations require expensive simulation. Furthermore, the relative efficiency of operations in prime fields is different from relative efficiency in a microprocessor's arithmetic logic units (ALUs). In addition to the expressiveness mismatch, this type and operator mismatch emphasizes the need for a new programming abstraction.

Domain-specific languages that compile to boolean or arithmetic circuits have been proposed in the context of secure multi-party computations (MPCs) (see Chapter 3). Computations have to be unrolled into oblivious circuits before execution. Yet, these languages are not suitable for translation into the ZoKrates intermediate representation: Since circuit-external witness derivation, as described in Chapter 7, is not supported, the cost for operations differs significantly. Again, this would lead to a type and operator mismatch, resulting in inefficient circuits.

While compilers in the context of hardware design, e.g., Verilog [323], VHDL [182], SystemC [181], also translate higher-level representations into boolean logic, they are not suitable in our context for several reasons: Resulting circuits are not purely combinational and cannot be represented through directed acyclic graphs (DAGs) as defined in Chapter 7. Instead, they are sequential, i.e., they have a notion of state represented through registers and contain loops, which allows re-use of subcircuits. Thus, they cannot be mapped to ZIR. Additionally, they take physical constraints into account, e.g., wiring constraints [230]. Most importantly, the established hardware description languages are quite different from programming languages software developers are used to, which conflicts with ZoKrates' usability goal.

In summary, we conclude that neither existing high-level programming languages, nor DSLs for MPCs or hardware designs are suitable high-level abstractions to specify verifiable off-chain computations.

Thus, we propose a new DSL, the ZoKrates language, which is designed to support developers

in specifying efficient and general off-chain computations in a usable way. While the ZoKrates language hides complexity associated with underlying verifiable computation schemes from its users, the compiler, which translates the language to ZIR, internally embraces and accounts for the idiosyncrasies of verifiable computation schemes and related abstractions to ensure maximum efficiency.

8.1.1 Domain-specific Challenges

The previous analysis showed that finding a suitable high-level abstraction for efficient verifiable off-chain computations comes with a unique set of challenges. Condensing the insights previously gained, we explicitly formulate these core challenges that are specific to our problem domain and the translation to the ZoKrates intermediate representation. This lays the foundation for the derivation of the ZoKrates language's design principles introduced in the next section.

- **Finiteness:** Programs have a fixed and finite number of inputs. The computational model is not uniform and not Turing-equivalent.
- **External Witness Derivation:** Efficient programs check external auxiliary inputs instead of directly computing them. For such programs, auxiliary inputs need to be computed during program execution, i.e., witness derivation.
- **Special Cost Model:** The cost of proving an off-chain program's correct execution depends on the number of constraints of its ZIR, i.e., the number of multiplication gates in the equivalent arithmetic circuit. The cost of program execution depends on all gates in that circuit, and the complexity of external witness derivation triggered through directives.
- **Practicality:** As shown in Chapter 7, any finite function's correct evaluation can be proved through a zk-SNARK. However, not every function evaluation can be proved correct in a time considered acceptable in practice. Hence, it is not a core challenge to construct a language that is as expressive as possible, but to ensure that programs specified in the language translate to ZIRs that are efficiently provable for problem sizes encountered in practice.
- **Confidentiality:** Off-chain computations need to support the confidentiality needs of the executing party, i.e., enable choice whether input information should become public with the proof of correct computation or remain private.
- **Developer Knowledge Gap:** Verifiable computation schemes are complex and their properties cannot be completely abstracted away from since data types and operators have a significant impact on efficiency. Yet, exposing developers with unfamiliar concepts may impair perceived usability and productivity.

8.1.2 Design Principles

To respond to these challenges, we define fundamental design principles that manifest in the ZoKrates language introduced in the subsequent section.

- **Imperative Programming Model:** Provide a simple imperative programming model that is accessible to blockchain developers, i.e., non-experts in the field of verifiable computations. Since Solidity [126], JavaScript [114], and many other established languages employed in the context of blockchain development follow this paradigm, iterative programming can be expected to be an easier, more familiar, and less error-prone model for program specification than arithmetic circuits.
- **Design Syntax for Efficiency:** Prevent developers from accidentally writing inefficient code by avoiding syntactic elements that are inherently inefficient in circuits. Design operators and data types accordingly. For example, field-native operations should be preferred over embedding boolean circuits whenever possible. In case there is an important but expensive primitive, we rather make it an importable standard library function (see Section 8.3) to raise awareness.
- **Transparency of Limitations:** Do not try to provide language features that cannot be efficiently provided. For example, being able to specify infinite loops only to be asked to give an upper bound of iterations at compile-time impairs usability.
- **Hide External Witness Derivation:** Do not expose external witness derivation to developers. While the program-external computation of witness information is a recurring theme, only a small set of essential solver functions is required (see Section 10.5). Hence, it is not necessary to support the specification of witness derivation logic in a high-level language. Doing so would add conceptual complexity without direct benefits. Instead, implement the necessary solver functions internally and insert directives to call solvers during translation to the ZIR as needed.
- **Public and Private Inputs:** Zero-knowledge verifiable computation schemes allow the witness to remain a secret to the prover. Allow developers to define input variables as either part of the proof or secret witness information.
- **Reliable Interpreter:** Every constraint in a given ZIR system needs to be satisfied for a witness to be valid. Thus, there can be no conditional, short-circuit evaluation in a high-level representation, even for conditional branches (e.g., if-else-statements). Conceptually, for the branches that do not satisfy a condition, arbitrary values for the corresponding witness values should be allowed. Asserting this, however, requires additional constraints and thus impairs proving efficiency (evaluation efficiency would benefit since an interpreter could select an arbitrary variable assignment for the corresponding witness variables).

  This cost overhead can be avoided by relying on the program's interpreter to derive a witness that fulfills the system even without additional constraints by evaluating all possible paths correctly. While this approach restricts the set of valid witnesses, it never changes a program execution's result. By taking this approach, proving efficiency is traded for efficiency of program execution, which is desirable since proving times dominate execution times in practice as described in Section 7.3.2.
- **Optimizing Compiler:** Optimize programs specified by developers during compilation, taking into account the domain-specific cost model. Support developers in writing ef-

ficiently provable programs by preventing the creation of unnecessary constraints and removing as many constraints as possible from the ZIR without compromising correctness in the compilation process.

8.2 Syntax and Semantics

Addressing the challenges previously defined by following the design principles, we introduce the ZoKrates Language in this section. We take an example-driven approach to introduce syntactic elements and their semantics. Furthermore, we explain how the elements of the high-level program specification in the ZoKrates language relate and translate to the ZIR.

The syntax of the ZoKrates DSL is loosely inspired by Python [330] but, due to the domain-specific requirements and peculiarities, deviates in many aspects and borrows concepts from various programming languages. For the formally inclined reader, we provide a formal syntax specification of the ZoKrates language in the form of a Parsing Expression Grammar (PEG) [136] in Appendix A.

8.2.1 Program Structure and Fundamentals

A ZoKrates program consists of at least one source code file with a `.zok` filename extension, which specifies a `main` function. Based on a simple example, we describe the basic structure of ZoKrates programs and fundamental concepts.

In this discussion of fundamentals, we occasionally use syntactic elements and concepts before they have been introduced. They are defined and described in more detail in later sections; we provide forward references.

A First ZoKrates Program

We provide a first ZoKrates Program in Listing 8.1.

The entry point of a ZoKrates program is the `main` function. This function can have public and private input arguments and return zero to many values representing the program's result.

A function's body comprises a list of statements, e.g., variable assignments or function calls. In the example given, two previously defined functions are called from the `main` function. The results are assigned to variables of types `field` and `bool`. These variables are then returned as the program's overall result. Variables are always passed by value, and there are no global functions. Thus, there can be no side-effects.

In the ZoKrates language, statements are newline-terminated; it does not require explicit statement separators (implicitly, this is `\n`). Indentations do not have semantic meaning but are purely aesthetic and support readability.

Listing 8.1: A First ZoKrates Program

```
/*
* ZoKrates Program Example
*/

// adds two numbers of type field
def add(field a,field b) -> (field):
  return a+b

// checks whether two field elements are equal
def equal(field a, field b) -> (bool):
  return a==b

// entry point for program execution
def main(private field a, public field b) -> (field, bool):
  field c = add(a,b)
  bool d = equal(a,b)
  return c, d
```

Public and Private Inputs

As depicted in the Listing 8.1, the `main` function's header allows the specification of optional visibility modifier for input parameters, which can have the values `public` or `private`. These modifiers are only available for the `main` function and have special semantics that defines how input information is processed when proving the program's correct execution: While public inputs become public information when a proof attesting the correct execution of the program is sent to the blockchain, the private inputs are not published and will remain the prover's secret.

Expressing this idea in terms of the ZoKrates intermediate representation, the visibility modifiers determine whether an input to a ZoKrates program maps to a ZIR input or an intermediate witness variable.

Identifiers

In the ZoKrates language, an identifier is the name of an entity, e.g., a variable, a function, or a struct.

Identifiers must start with a letter. Thereafter, they can contain arbitrary combinations of letters, numbers and underscores. An identifier must not be a reserved keyword, but is allowed to start with or contain a keyword. In the ZoKrates Parsing Expression Grammar [136], a valid identifier is expressed as follows:

```
identifier = @{ ((!keyword ~ ASCII_ALPHA) | (keyword ~ (ASCII_ALPHANUMERIC
    | "_"))) ~ (ASCII_ALPHANUMERIC | "_")* }
```

The ZoKrates language reserves the following 24 keywords that can not be used as identifiers:

```
as        bool      def       do        else
endfor    export    false     field     for
from      if        then      fi        import
in        private   public    return    struct
true      u8        u16       u32
```

Scope

The scope of an identifier describes the region in the source code where the identifier is visible and accessible. For a given point in a program, an identifier is *in scope* if it can be used to validly refer to the entity it identifies.

ZoKrates is statically scoped, i.e., an identifier's scope is determined at compile-time and only depends on the program's source code (as opposed to its state during execution as in dynamic scoping).

Function scope is the primary local scoping mechanism in the ZoKrates language and global scope is not supported. A function definition creates a new scope which comprises all identifiers declared therein. Consequently, these identifiers move out of context when another function is called. As nested control flow elements, loops have their own scope, which includes variables defined in their headers. There is an exception to these basic scoping rules: There exists a dedicated module scope that can be extended through imports so that functions and user-defined structs defined in other modules can be imported (see Section 8.2.5).

Re-declarations of variables previously declared within a scope are not allowed. This rule extends to nested scopes: Shadowing of variables, i.e., re-declaration of variables declared in an outer scope within an inner scope, is not permitted. Together, these rules ensure that an identifier always uniquely refers to the same variable across nested scopes within a module.

In Listing 8.2, we provide an example program comprising a set of statements that comply with or violate scoping rules.

Comments

Like most programming languages, the ZoKrates language supports inline and block comments. Both types of comments can be observed in Listing 8.1.

Inline comments can be introduced by `//` and mark the remainder of the line as an uninterpreted comment, i.e., the comment is implicitly closed through a newline character (`\n`).

Block comments, in contrast, can span multiple lines; beginning and end are explicitly specified. A block comment is opened with `/*` and closed with `*/`.

8.2.2 Types

In this section, we define data types supported by the ZoKrates language. There are three *primitive types*, which are types that directly map to field elements internally: `field`, `bool`, and unsigned

Listing 8.2: ZoKrates Scoping Examples

```
from "./foo_module" import foo // brings function 'foo' into module scope

// bool t = true <- global variable not allowed

def bar(field a) -> (field):
    return a

def main() -> ():
  field a = 2
  // field a = 3 <- re-declaration not allowed

  field c = bar(a)
  bool d = foo(a,true)

  for field i in 0..5 do
      // field a = 7 <- shadowing not allowed
      a = 7
      field b = i
  endfor
  // field e = b <- out of loop-scope access not allowed

  return
```

integers (`u8`, `u16`, `u32`). Furthermore, there are two *composite types*, which combine multiple primitive types into more complex data structures: Instances of the same type can be grouped in — optionally multidimensional — *arrays*, whereas more complex user-defined compositions of various types — including nesting — can be expressed through *structs*.

In ZoKrates, variable declarations of any type always require initialization, i.e., an assignment, in the same statement. Listing 8.3 shows an example program which contains declarations of primitive types, including initialization with literals of the respective types.

Listing 8.3: Primitive ZoKrates Type Declaration and Initialization

```
def main() -> (field, bool, u32):
    field a = 1
    bool b = false
    u32 c = 0x11133777
    return a,b,c
```

Field Element

The most basic data type in ZoKrates is a field element. This data type is indicated as `field`. This data type directly exposes the only data type of the ZoKrates intermediate representation, a prime field element, i.e., a positive integer modulo a fixed prime number. The developer can think of a prime field element as a simple unsigned integer type of high bitwidth in most cases. In

our implementation instantiated for Ethereum, this is 254 bits, the prime field modulus' bitwidth (see Chapter 10).

Boolean

The boolean type is the second primitive data type supported by the ZoKrates language. It is indicated by the `bool` keyword. A variable of type boolean has only two possible values; we define the literals representing these values to be `{true, false}`. Booleans are commonly used within conditional statements and are the result type of logical expressions.

From the perspective of the ZoKrates intermediate representation, booleans are purely syntactic sugar and internally represented as field elements with values 0 or 1. To guarantee correctness, such field elements need to be constrained to these two values within the translation process. We can force a field element a to be in $0, 1$ by adding the following constraint to the ZIR:

```
a*(1-a)==0
```

Unsigned Integers

Unsigned integers represent the third primitive type in ZoKrates. There are three different instantiations of the unsigned integer type; 8, 16, and 32 bits. Unsigned integer data types consist of a leading `u` and a postfix denoting the bitwidth, i.e., `u8`, `u16`, and `u32`.

In the ZIR, unsigned integers are usually represented as field elements for efficiency. During translation, the ZoKrates compiler tracks potential overflows and forces a conversion to a variable's binary representation only when needed, e.g., when a result needs to be truncated, or when a bitwise operator is applied.

Array

ZoKrates offer *arrays* as a complex data type that allows grouping values of the same type in one data structure. The values of an array are referred to as its elements. Elements can be of any type supported by ZoKrates. This includes arrays themselves, which directly implies support for multidimensionality. Arrays in ZoKrates are static, i.e., their length needs to be known at compile-time so that translation to ZIR is possible. Indexing is zero-based. Before discussing this translation in more detail, we describe how arrays are declared, initialized, and used, as well as special syntax for their manipulation.

Declaration and Inititalization: A one-dimensional array of elements of a given type can simply be declared by appending square brackets `[]` to the identifier of a variable declaration for that type. As previously explained, a declaration always requires an initialization within the same statement in ZoKrates. An array's size is specified in the square brackets, as shown in the following statement:

```
field[3] a = [1, 2, 3] // initialize a field array with field values
```

ZoKrates offers a special shorthand notation for the initialization of an array with elements of constant value. For this, the constant value and the number repetitions are provided in square brackets: `[value; repetitions]`. This notation is especially useful when initializing large arrays, for example:

```
bool[256] b = [0;256] // initialize a boolean array with 0s
```

Multidimensional arrays are essentially arrays of arrays of elements of a type. For a pair of nested arrays in such a structure, we call the array that contains the other array the outer array. Analogously, we refer to the contained array as *inner array*. To declare an array of an inner array, i.e., an outer array of inner array elements, prepend brackets [size] to the size definition of the inner array. For a two-dimensional example, the leads to the following declaration and initialization statement:

```
// outer array has size 2, inner array size 3
field[2][3] a = [[1, 2, 3],[4, 5, 6]]
```

Generalizing this example leads to the following rule for multidimensional array declarations:

```
data_type[size of 1st dim.][size of 2nd dim],...,[size of nth dim]
```

Syntactic Sugar: In ZoKrates, there are two dedicated operators that make working with arrays syntactically less verbose and more convenient for developers: the spread and the slice operator.

The *spread* operator, denoted `...`, is a unary prefix operator which copies elements of an existing array it is applied to. This is especially useful when deriving new arrays from existing ones, for example:

```
field[3] a = [1, 2, 3]
field[4] c = [...a, 4] // initialize an array by copying values from "a" and
    appending "4"
```

The *slice* operator is a binary postfix operator that creates a new array with values from a slice of an existing array. A slice is created from an existing array a by specifying a start index i and an end index j in square brackets: `a[i..j]`. Hereby, the slice indices must respect the size of the array from which the slice is derived. The start index is inclusive; the end index is exclusive. This operator is especially useful when subsets of existing arrays should be used in the context of parameters or initialization as in the following example:

```
field[3] a = [1, 2, 3]
field[2] b = a[1..3]   // initialize an array copying a slice from `a`
```

Examples: We provide an example ZoKrates program that summarizes and showcases how arrays can be declared, used, and manipulated in Listing 8.4.

Listing 8.4: Usage Examples for the ZoKrates Array Type

```
1 def main() -> (field):
2     field[3] a = [1, 2, 3] // initialize a field array with field values
3     a[2] = 4               // set a member to a value
4     field[4] b = [42; 4]   // initialize an array of 4 values all equal to 42
5     field[4] c = [...a, 4] // initialize an array copying values from `a`,
          followed by 4
6     field[2] d = a[1..3]   // initialize an array copying a slice from `a`
7     bool[3] e = [true, true || false, true] // initialize a boolean array
8     field[2][3] f = [[1, 2, 3],[4;3]] // initialize a two-dimensional array
9     return a[0] + b[1] + c[2] + f[1][2]
```

Translation: As explained in Section 7.4, there is no notion of arrays in the ZoKrates intermediate representation, but only variables representing field elements. Thus, arrays need to be eliminated during translated into the ZIR.

For one-dimensional arrays, this process is straight-forward: An array is resolved by converting array elements to separate variables of the contained element's type. Thereafter, the translation of these variables to field elements happens in a type-dependent way. The variable associated with the access of an array element at an index could be determined through its offset in the list of variable declarations. A simpler way of identifying a variable associated with access at an index is to encode the index in its identifier. Consider the following example:

```
field[64] a = [16;64]
field b = a[31]
```

In the ZoKrates intermediate representation, the assignment in the second line can be represented through the following constraint using index-based identifiers:

```
b == a_31
```

For multidimensional arrays, the same transformation is applied recursively until the array has been unrolled into variables of the type of the innermost array's elements. Intuitively, this translation can be understood as declaration of intermediate variables with names derived from the variables' position in an array for all dimensions, i.e., an extension of the previously introduced index-based variable naming. Consider the following example of a three-dimensional array:

```
field[2][2][3] a = [[[1, 2, 3],[4, 5, 6]],[[7, 8, 9],[10, 11, 12]]]
field b = a[1][0][2] // 9
```

The following ZIR-constraint represents the assignment in the second line:

```
b == a_1_0_2
```

Struct

A *struct* is a user-defined composite data type that is formed by a named collection of variables. The contained variables can be of any type. Nesting of structs is supported. Before a struct can be used in the definition of another struct, it needs to be defined. This rule prevents the specification of unresolvable cyclic references in struct definitions.

Definition: Before the struct data type can be used in declarations and assignments, it needs to be defined. A struct definition starts with the 'struct' keyword followed by a name. Afterward, a newline-separated list of variables, so-called members, is declared in curly braces `{}`. The syntax for member declaration follows the regular syntax for variable declarations. Note, however, that initialization of these variables is not required within struct definitions.

The following example defines a struct representing a point which consists of two members of type field. These members represent the point's coordinates:

```
struct Point {
  field x
  field y
}
```

Declaration and Inititalization: Like all declarations in the ZoKrates language, the declaration of a variable of a struct type needs to happen in the same statement as its initialization (unless the struct-typed variable is declared within a function's signature).

To support initialization, ZoKrates supports a special inline syntax for the definition of struct instances. A comma-separated list of `name:value` pairs, where the name identifies a member, is given in curly braces:

```
{member: value [, member: value]}
```

Note, that a value can be a struct instance again. Thus, this notation directly supports nested structs. The following example shows declaration and initialization of a variable of the previously defined `Point` struct type:

```
def main() -> (Point):
  Point p = Point {x: 1, y: 0}
  return p
```

Accessing Members: Members of a struct type can be accessed through the dot (`.`) operator. Herefore, the member's name is appended to a variables identifier of type struct separated by the dot. The following example shows how struct members can be accessed using this operator:

```
p.x = a // assign a to x coordinate
p.y = p.x // assign x coordinate to y coordinate
```

Examples: In Listing 8.5, we provide a slightly more complex program that exemplarily shows the usage of structs in ZoKrates.

Listing 8.5: Example ZoKrates Program Using the Struct Data Type

```
1  struct Bar {
2      field[2] c
3      bool d
4  }
5
6  struct Foo {
7      Bar a
8      bool b
9  }
10
11 def main() -> (Foo):
12     Foo[2] f = [Foo { a: Bar { c: [0, 0], d: false }, b: true}, Foo { a: Bar
           {c: [0, 0], d: false}, b: true}]
13     f[0].a.c = [42, 43]
14     return f[0]
```

Translation: Like arrays, structs cannot be represented in the ZoKrates intermediate representation, but need to be flattened, i.e., converted into a set of ZIR-constraints that only contain field elements. This process is similar to the approach for the array data type:

Structs are resolved during translation by representing its members as a list of typed variables that can be accessed through their offsets. This process happens recursively until all nested structs have been resolved. Thereafter, translation of the member variables to field elements happens as defined for each member's type.

8.2.3 Operators

In this section, we describe the operators available in the ZoKrates language, their meaning, and how they can be combined.

Operators link identifiers and constants to form expressions. The evaluation of an expression returns a typed result. Expressions can not have side effects, as they do not comprise assignments.

Arithmetic Operators

Table 8.1 provides an overview of the *arithmetic operators* available in the ZoKrates language. With the exception of the modulo operator, which is only available for unsigned integer types, all operators are defined for the `field` and unsigned integers. All arithmetic operators are binary

operators that map a pair of values of a type to a single value of that type, e.g., $\mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$. Operations have positive integer semantics unless there are overflows with the exception of division with remainder for field elements (see Section 7.1).

Table 8.1: Arithmetic Operators

Operator	Description	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`%`	Modulo	`x % y`
`**`	Power	`x**c`

The translation of arithmetic operators to the ZoKrates intermediate representation is trivial in the case of multiplication and addition: The operators are directly available in the ZIR. As described in depth in Section 7.3.2, the division operator is translated to a multiplication operation and an external witness derivation directive. For field elements, this directive finds the divisor's multiplicative inverse element. The subtraction operator can be translated by additive inversion of the scalar preceding a variable. For unsigned integers, the directive deconstructs division result in an integer division part and a remainder. In the ZIR, we assert that the deconstruction is valid, does not overflow, and the remainder is in range.

A noteworthy restriction applies to the power operator: The power operator is translated to a series of additions and multiplications in the ZIR. For efficient exponentiation, a square-and-multiply algorithm [205] is used instead of merely multiplying the base with itself repeatedly. As each multiplication requires one ZIR-constraint and the number of constraints needs to be finite and known at compile-time, the exponent has to be a constant.

Comparison Operators

An overview of the *comparison operators* available in the ZoKrates language is given in Table 8.2. We distinguish two groups of comparison operators based on the type an operator is defined for:

1. The *equality operators*, == and `!=`, are defined for all primitive types. They map two elements of their input type to a boolean value. For field elements, for example, these binary operators compute a function $f_{eq}\colon \mathbb{F}_p \times \mathbb{F}_p \to \{0,1\}$, for booleans they compute a function $f_{eq}\colon \{0,1\} \times \{0,1\} \to \{0,1\}$.

2. The *relational operators*, <, <=, >, and >=, are defined only for the `field` and unsigned integer types. They map two elements of the input type to a boolean value. Hence, these binary operators compute a function $f_{rel}\colon \mathbb{F}_p \times \mathbb{F}_p \to \{0,1\}$ and $f_{rel}\colon \mathbb{N}_{uint} \times \mathbb{N}_{uint} \to \{0,1\}$, where $\mathbb{N}_{uint} = \{x \in \mathbb{N} : 0 \leq x < 2^{bitwidth}\}$ respectively.

Table 8.2: Comparison Operators

Operator	Description	Example
==	Equal	`x == y`
!=	Not equal	`x != y`
<	Less than	`x < y`
<=	Less than or equal	`x <= y`
>	Greater than	`x > y`
>=	Greater than or equal	`x >= y`

Equality operators could be implemented by decomposing their operands into binary representations and then performing bit-level comparisons. However, there is a more efficient translation that uses native field operations without requiring embedding of binary logic [303]: Let `x!=y` be an expression in the ZoKrates language, where the operands are variables of a primitive type. According to the != operator's definition, this expression evaluates to `bool`.

Translating this problem into the ZoKrates intermediate representation, we search for an arithmetic expression that evaluates to result variable r and has value 0 in case of equality and 1 in case of inequality of the field element operands x and y. Recall, that `bool` literals map to $0, 1$ in the ZIR. Formally, we want to construct an expression that evaluates to r. Additional constraints for correctness are acceptable:

$$r = f_{\neq}(x, y) = \begin{cases} 1 & x \neq y \\ 0 & x == y \end{cases} \tag{8.1}$$

By definition, we know that any field element but 0 has a multiplicative inverse (see Section 7.1). We can therefore define the following equation which links the inequality operator with an arithmetic expression's satisfiablilty:

$$\begin{aligned} & x \neq y \\ & x - y \neq 0 \\ \iff & \exists w \in F_p : w * (x - y) = 1 \\ \iff & \exists w \in F_p : w * (x - y) - 1 = 0 \end{aligned}$$

Using this insight, we can now define an arithmetic expression for r as defined in Equation (8.1). This expression alone is not sufficient, it merely ensures that $r = 0$ for $x = y$ and $r \neq 0$ else. Thus, an additional constraint forces r to be 1 in case $x \neq y$.

$$\begin{aligned} & r = f_{\neq}(x, y) = \begin{cases} 1 & x \neq y \\ 0 & x == y \end{cases} \\ \iff & r = w * (x - y) \wedge (1 - r) * (x - y) = 0 \end{aligned} \tag{8.2}$$

In case $x = y$, any value for w satisfies the constraints. For $x \neq y$, however, w must be the multiplicative inverse of $(x - y)$ for the constraints to hold. Since finding a multiplicative inverse is inefficient in arithmetic circuits (see Section 7.3.2), we rely on external witness derivation. Before inserting the inequality constraints derived in Equation (8.2), we insert a directive that retrieves $(x - y)^{-1}$ into the ZIR.

The equality operator == can simply be translated to the ZIR by using the previous result and inverting the boolean result:

$$x = y \iff (1 - (x \neq y)).$$

Unlike the equality operators, relational operators cannot be directly translated to ZIR-native field arithmetics, but require their operands to be decomposed into their binary representation (see Chapter 7). Consider the $<$ operator with operands $x, y \in \mathbb{F}_p$. We look for an expression that evaluates to result $r \in \{0, 1\}$:

$$r = f_<(x, y) = \begin{cases} 1 & x < y \\ 0 & x \geq y \end{cases} \tag{8.3}$$

First, bit decompositions $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1} \in \{0, 1\}$ of x and y are found through external witness derivation with $n = bitwidth(|\mathbb{F}_p|) = \lceil log_2(p) \rceil$ so that:

$$x = \sum_{i=0}^{n} x_i * 2^i, \qquad y = \sum_{i=0}^{n} y_i * 2^i$$

Hereby, uniqueness of the decomposition needs to be ensured as described in Section 7.3.2. Then, a standard combinational comparator circuit $C_<$ [234] for $n = \lceil log_2(p) \rceil$ bits is applied to the decompositions. It computes $C_<\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. Applied to the bit decompositions of x and y, the comparator circuit evaluates to $r = C_<((x_0, \ldots, x_{n-1}), (y_0, \ldots, y_{n-1}))$ as described in Equation (8.3). Restricting the operands of the $<$ operator to a bitwidth $\lceil log_2(p) \rceil - 2$ bits allows for a construction that only requires one binary decomposition and is thus more efficient [303]. While this restriction does not generally apply to the $field$ type, it is leveraged in the case of unsigned integers.

Given the $<$ operator, the $>$ operator is trivial to implement through swapping operands:

$$x > y \iff y < x$$

As proved in Table 8.3, the $>=$ can be expressed using $<$. The same transformation can be applied for $<=$, which can be expressed using $>$:

$$x \geq y \iff 1 - (x < y), \qquad x \leq y \iff 1 - (x > y)$$

While expressing $x \leq y$ as $(x < y) \wedge (x = y)$ is also correct, it requires more constraints in the ZIR and is therefore avoided.

Table 8.3: Equivalence Proof for Relational Operator Transformations

x	y	$x < y$	$x \geq y$	$1 - (x < y)$	$x > y$	$x \leq y$	$1 - (x > y)$
1	1	0	1	1	0	1	1
1	0	0	1	1	1	0	0
0	1	1	0	0	0	1	1
0	0	0	1	1	0	1	1

Logic Operators

Table 8.4 gives an overview of the *logic operators* provided by the ZoKrates language. These operators are only defined for the `bool` data type. The binary operators, `&&` and `||`, map two boolean operands to a boolean result, i.e., they compute a function $f_{logic}\colon \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$. The unary `!` operator maps a boolean operand to a boolean result, which can be written as a function $f_{logic}\colon \{0,1\} \rightarrow \{0,1\}$.

Table 8.4: Logic Operators

Operator	Description	Example
`!`	Not	`!x`
`&&`	And	`x && y`
`||`	Or	`x || y`

In the translation from the ZoKrates language to the ZoKrates intermediate representation, boolean operands `x,y` $\in$ `{true|false}` are transformed to field elements $x, y \in \{0,1\}$. Logic operators themselves can be transformed by expressing them trough arithmetic operations as shown in Table 8.5. We prove equivalence of these arithmetic expressions in Table 8.6.

Table 8.5: Logic Operations as Arithmetic Expressions

Operation	Arithmetic Expression
`!x`	$1 - x$
`x && y`	$x * y$
`x ^ y`	$x + y - 2(x * y)$
`x || y`	$1 - (1 - x) * (1 - y)$

Bitwise Operators

Bitwise operators are only defined for unsigned integers and directly operate on their binary representation. Table 8.7 provides an overview of the bitwise operators available in the ZoKrates language.

Table 8.6: Equivalence Proof for Logic Operator Transformations

x	$!x$	$1-x$
1	0	0
0	1	1

x	y	$x\&\&y$	$x*y$	$x\|\|y$	$1-(1-x)*(1-y)$	$x\hat{}y$	$x+y-2xy$
1	1	1	1	1	1	0	0
1	0	0	0	1	1	1	1
0	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0

The translation of bitwise operators trivially reduces to the application of the previously defined boolean operations on bits of the binary representations of operands.

Table 8.7: Bitwise Operators

Operator	**Description**	**Example**
`<<`	Right Shift by Constant	`x << 1`
`>>`	Left Shift by Constant	`x >> 2`
`&`	Bitwise And	`x & 0xf0`
`|`	Bitwise Or	`x | y`
`^`	Bitwise Exclusive Or	`x ^ y`

Miscellaneous Operators

The group of *miscellaneous operators* contains all operators defined in the ZoKrates language that do not fit into any of the categories above. Table 8.8 provides an overview of these operators. Operators in this group bind to identifiers more strongly than those of any other group and are evaluated first in any expression (they take precedence).

Table 8.8: Miscellaneous Operators

Operator	**Description**	**Example**
`()`	Function Call	`x(y)`
`[]`	Array Access	`x[3]`
`[..]`	Slice	`x[1..5]`
`...`	Spread	`...x`
`.`	Struct Member Access	`x.y`

All operators in the miscellaneous category and their translation to the ZoKrates intermediate

representation have already been introduced in previous sections.

Precedence Rules and Associativity

The operators introduced in previous sections can be combined to form arbitrarily complex expressions. Herein, explicitly defining evaluation order by grouping sub-expressions in parentheses `()` is always possible. Nonetheless, a sensible evaluation order for expressions comprising multiple operators should exist without explicit definitions.

Therefore, we define precedence rules and associativity for the operators of the ZoKrates language in Table 8.9. An operator of higher precedence is evaluated before an operator with lower precedence. In Table 8.9, a horizontal line separates a group of operators of equal precedence, and the priority of groups of operators decreases from top to bottom. In case a sub-expressions comprises operators of equal precedence, the associativity decides on the evaluation order.

Table 8.9: Precendence Rules and Associativity of ZoKrates Operators

Operator	**Description**	**Associativity**
()	Function Call	Left to Right
[]	Array Access	
.	Struct Member Access	
[..]	Slice	
...	Spread	
!	Not	Right to Left
**	Power	Left to Right
*	Multiplication	Left to Right
/	Division	
%	Modulo	
+	Addition	Left to Right
-	Subtraction	
<<	Right Shift by Constant	Left to Right
>>	Left Shift by Constant	
&	Bitwise And	Left to Right
|	Bitwise Or	Left to Right
^	Bitwise Exclusive Or	Left to Right
<	Less than	Left to Right
<=	Less than Equal	
>	Greater than	
>=	Greater than Equal	
==	Equal	Left to Right
!=	Not Equal	
&&	And	Left to Right
||	Or	Left to Right

8.2.4 Control Flow

In this section, we describe language elements that can be used to define and influence the control flow of ZoKrates programs. *Functions* encapsulate functionality and can be invoked from anywhere in a program. Code within *loops* is repeatedly executed and *conditionals* enable branching.

Functions

In the ZoKrates language, a function has to be defined before it can be called. This rule prevents unsupported recursive calls by design. All function arguments are always passed by value, so that no side effects can occur. Functions have their own static scope, which comprises everything between the function's header and the return statement.

Definition: A function definition has the following structure:

```
def identifier(parameters) -> (return_types):
function_body
return return_values
```

The first line represents the function header. In the definition,

- a function's *identifier* can be any valid ZoKrates language identifier.
- *parameters* is a comma-separated list of variable declarations. A function is not required to have arguments.
- *return_types* specifies the types of the values returned by the function as comma-separated list that can be empty.
- the values returned by the function are specified as a comma-separated list in *return_values*. This list can be empty.
- the *function_body* consists of a list of statements that define the function's logic.

ZoKrates function definitions can be overloaded, i.e., the same identifier can be used in multiple function definitions. Uniqueness needs to be ensured for function signatures, the combination of identifier, parameter types, and return types. Signature uniqueness guarantees that a function call always unequivocally maps to a definition. The mapping of function calls to definitions is exact; there are no implicit typecasts.

Invocation: A function can be called by applying the function call operator `()` to its identifier:

```
identifier()
```

Example: We provide an example ZoKrates program that contains function definitions and invocations in Listing 8.6.

Listing 8.6: ZoKrates Functions Example

```
def foo(field x) -> (field, field[3]):
    return x, [2, 3, 4]

def foo(field x) -> (field, field):
    return 1, x

def main(field x) -> (field):
    a, field[3] b = foo(x)
    return 1
```

Translation: The concept of functions, or even blocks of code, does not exist in the ZoKrates intermediate representation. Thus, function calls need to be resolved in the translation process. Resolution happens through inlining; a technique commonly applied as an optimization in compiler design [4]: A function call is replaced by the called function's body for all occurrences. Function call inlining happens recursively to resolve nested function calls within definitions.

Assertions: There exists a special predefined *assert* function in ZoKrates. This function accepts a boolean expression as its argument and asserts that the condition always holds. An assertion is directly translated to a set of ZIR constraints.

```
    assert(boolean_expression)
```

Loops

The ZoKrates language supports for-loops to specify instructions to be performed repeatedly. Since loops have to be unrolled into a finite list of statements during the compilation process (refer to Section 8.4) to the ZoKrates intermediate representation, the number of iterations needs to be bounded. Thus, loop bounds need to be known at compile-time. For the same reason, recursion is not supported. The body of a for-loop has its own scope, which comprises the loop counter variable.

Definition: A for-loop has the following structure:

```
    for field identifier in counter_init..upper_bound do
      loop_body
    endfor
```

In this definition,

- the *identifier* defines the name of the loop counter variable of type `field` that tracks the number of iterations. This variable is in the loop's scope.
- the loop counter variable is initialized with the *counter_init* value of type `field`.

- the *upper_bound* value of type `field` defines an exclusive upper bound for the loop counter variable.
- the *loop_body* consists of a list of statements that are evaluated in every iteration.

Examples: We provide an example of a ZoKrates program illustrating the use of a for-loop in Listing 8.7.

Listing 8.7: ZoKrates For-Loop Example

```
1 def main() -> (field):
2     field res = 0
3     for field i in 0..7 do
4         res = res + i
5     endfor
6     return res
```

Translation: As derived in Chapter 7, the ZoKrates intermediate representation can not contain loops or jumps, but rather represents a straight-line program. Therefore, loops have to be *unrolled* during translation: A loop is replaced by the loop's body number-of-iterations–times, resulting in a linear sequence without jumps. Unrolling happens recursively to resolve loops within loops. Albeit with a different goal, loop-unrolling is also used as an optimization technique in traditional compiler design for Turing-complete abstractions [4].

The previously introduced for-loop syntax guarantees compile-time constant loop bounds and thereby ensures that unrolling is always possible. Developers cannot specify loops that cannot be unrolled. For the same reason, we choose not to support other common loop types, e.g., while- or repeat-until-loops. Their entry and exit conditions would have to evaluate compile-time constants — an unusual constraint not directly enforceable through syntax and behavior most likely unexpected by a developer.

Conditionals

The ZoKrates language provides conditionals in the form of if-then-else expressions, i.e., expressions that have different results depending on a condition's logical value. These if-then-else-expressions enable conditional assignments. Conditional blocks are not supported in the ZoKrates language. While this concept could be implemented and mapped to the ZIR, it would lead to undesirable constraint overhead for syntactic sugar.

Definition: An if-then-else expression has the following structure:

```
    if condition then if_expr else else_expr fi
```

In this definition,

- the *condition* is an expression that evaluates to a value of type `bool` and encodes the condition the result of the if-then-else expression depends on.

- *then_expr* and *else_expr* are expressions that have to evaluate to values of the same type. If the condition evaluates to `true`, the if-else-then expression evaluates to then_expr and to else_expr otherwise.

Examples: We provide an example of how conditionals can be used in a ZoKrates program in Listing 8.8.

Listing 8.8: ZoKrates Conditionals Example

```
1 def foo(field x) -> (field):
2   return if x/2==1 then 0 else x**3 fi
3
4 def main(field x) -> (field):
5   field y = 2 * if x + 2 == 3 then 0 else foo(x) fi
6   return y*x
```

Translation: As part of the description of comparison operators supported in the ZoKrates language, we previously showed how conditional expressions translate to the ZoKrates intermediate representation so that they evaluate to a result variable $r \in \{0, 1\} \subset \mathbb{F}_p$. Building on this insight, the translation of an if-then-else expression becomes trivial. We know that the condition evaluates to $\{0, 1\}$. Hence, we can simply express the if-then-else expression as a linear combination:

```
condition * then_expr + (1-condition) * else_expr
```

Note, that both expressions, then_expr and else_expr, are evaluated in any case. There is no short-circuit evaluation. Thus, ZoKrates programs cannot be optimized through conditional blocks. For this reason, offering conditional blocks in the ZoKrates language would be misleading and purely syntactic sugar.

8.2.5 Modules and Imports

A ZoKrates module is a syntactically valid ZoKrates file with a `.zok` filename extensions. It does not have to contain a `main` function.

Import statements allow entities to be imported from such modules and brought into the importing module's scope. Hereby, an importable *entity* can be a function or a struct definition. The *module scope* is a scope that contains all structs and functions defined in or imported to a module.

Definition: An import statement has the following structure:

```
from path import identifier as local_identifier
```

Hereby,

- the *path* is a path string, enclosed in quotation marks `""`. It specifies the location of a module to be imported from the filesystem. Hereby, the `.zok` filename extension is optional. There are three types of paths:

  1. A *relative path* specifies the imported module's location relative to the importing module. Thus, the path string starts with the working directory selector `"./"`.

  2. An *absolute path* specifies the imported module's location in a way that is not dependent on the importing module's location. An absolute path starts with the root directory selector `"/"`.

  3. A path that does not start with one of these selectors is evaluated relatively to the root directory of the ZoKrates standard library (see Section 8.3) to conveniently import entities from contained modules.

- The *identifier* determines which entity, i.e., function or struct definition, shall be imported from the referred module. The *local_identifier* specifies an alias for this entity, which is valid within the importing module's module scope. The specification of a local identifier, including the `as` keyword, is optional. By default, the filename (without extension) of the imported module is used.

Examples: We provide an example program which imports ZoKrates modules in Listings 8.9 to 8.12.

Listing 8.9: ZoKrates Modules and Imports Example

```
1  from "./foo.zok" import main as foo // absolute import of main function from
       module "foo" with local identifier "foo"
2  from "./bar" import main      // relative import of main function from module
       "bar" with module identifier as local identifier
3  from "./baz.zok" import qux as baz // the .zok file ending is optional
4  from "hashes/sha256/512bitPacked" import main as sha256 // imports sha256
       from standard library
5  from  "./point.zok" import Point as Point // import of "Point" struct
6
7  def main(field x) -> (field[2]):
8      Point p = Point{x:2,y: x + bar() + baz()}
9      field[2] result = sha256(foo(p.y)) // use the imported functions
10     return result
```

Listing 8.10: ZoKrates Module `bar.zok`

```
1  def main() -> (field):
2    return 21
```

Listing 8.11: ZoKrates Module `foo.zok`

```
1 def main(field a) -> (field[4]):
2   return [a, 2*a, 4*a, 8*a]
```

Listing 8.12: ZoKrates Module `point.zok`

```
1 struct Point {
2     field x
3     field y
4 }
```

Translation: In the process of translating the ZoKrates language to the ZoKrates intermediate representation, imports are resolved by copying all referenced entities into the importing module. This resolution happens recursively to process nested imports in imported modules.

8.3 The ZoKrates Standard Library

The ZoKrates standard library is an extensible collection of useful functions written in the ZoKrates language itself. It offers a set of functions to support common domain-specific tasks, e.g., hashing and digitally signing, in an efficient way. Furthermore, it comprises utility functions that support developers in creating their own modules.

In this section, we highlight particularly interesting and domain-specific aspects of the standard library. By design, the standard library is extensible and our description is not intended as comprehensive documentation. Thus, we do not describe and discuss all functionality provided in-depth.

8.3.1 Utility Functions

The standard library provides several utility functions that support developers with common tasks encountered when writing programs in the ZoKrates language. Such common tasks are, for example:

- **Packing & Unpacking:** *Packing* describes the transformation of a value in binary representation to a representation as a field element. The inverse operation, *unpacking*, decomposes a field element value into its binary representation. Both operations are common tasks, e.g., when implementing algorithms that are defined for binary data such as cryptographic hash functions.

  As explained in Section 7.3.2, the unpacking operation is not necessarily injective, a behavior that may not expected by developers. To lift the burden of explicitly handling such cases, the unpacking functions we provide guarantee the uniqueness of outputs unless they are explicitly marked as `non-strict`.

- **Lookups:** Selecting from a set of pre-computed results instead of computing them within a ZoKrates program can reduce the number of ZIR-constraints considerably. For this purpose, the availability of *lookup* or multiplexing operations with little overhead is crucial. The standard library provides lookup functions, which support developers in efficiently selecting elements from arrays.
- **Casting:** The conversion between different data types, e.g., converting a `u32` variable to a `field` element, is supported through a set of typecasting functions.

8.3.2 Elliptic Curve Cryptography

Many modern cryptographic protocols rely on algebraic groups for which the discrete logarithm problem [295] is assumed to be hard as basic building blocks, for example, Diffie-Hellman key exchange [100] or Schnorr signatures [296]. Elliptic curves defined over finite fields form such a group and allow particularly efficient implementations of such protocols, e.g., elliptic curve Diffie-Hellman key exchange (ECDH) [26], or the elliptic curve digital signature algorithm (ECDSA) [191].

To support efficient cryptographic operations in ZoKrates programs, an implementation of elliptic curves is desirable. However, standard binary implementation approaches of elliptic curve group operations would be extremely inefficient and impractical. Fortunately, as subsequently explained, an elliptic curve can be sampled so that it only requires ZoKrates-native field arithmetics for its group operation. The resulting curve's group operation can thus easily be implemented in the ZoKrates language and translated to comparatively few ZoKrates intermediate representation constraints.

A twisted Edwards elliptic curve [43] over a non-binary field $\mathbb{F}$ is defined through the following equation fulfilled by curve points $(x, y) \in \mathbb{F} \times \mathbb{F}$.

$$E_{E,a,d} : ax^2 + y^2 = 1 + dx^2y^2$$

with curve parameters $a, d \in \mathbb{F}$. The field $\mathbb{F}$ is referred to as the curve's base field.

The addition law of two points $(x_1, y_1), (x_2, y_2)$ on a twisted Edwards curve is defined as:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

For a twisted Edwards curve over a finite prime field $\mathbb{F}_q$, the addition law can be expressed as:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right) \pmod q$$

In ZoKrates, the `field` data type represents elements of a prime field $\mathbb{F}_p$ (see Chapter 7). Thus, by setting this prime field $\mathbb{F}_p$ as base field $\mathbb{F}_q$ of the twisted Edwards curve, we would obtain a curve with an addition law that is efficiently computable in prime field arithmetics since $p = q$. Langley et al. provide an algorithm for generating parameters for such an elliptic curve over a prime field $\mathbb{F}_p$ for a given prime p [216].

Using this algorithm, Jubjub [56, 177], a twisted Edwards curve over the scalar field of the BLS12-381 elliptic curve, was generated in the context of Zcash's sapling protocol [364]. However, as explained in Chapter 7, the Ethereum Virtual Machine only supports the BN254 curve, which has a different order scalar field. Therefore, the JubJub curve cannot be used in the context of ZoKrates, where compatibility with Ethereum needs to be ensured. BN254's scalar field has the order of the BN254 curve's large prime-order subgroup, i.e., $p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

To support this specific scalar field, another curve called BabyJubJub was sampled following the same approach [345, 346]. The resulting twisted Edwards curve fulfills the SafeCurve security criteria [44] and has parameters $a = 168700$, and $d = 168696$, resulting in the following curve equation:

$$E_{E,BabyJubJub} : 168700x^2 + y^2 = 1 + 168696x^2y^2$$

Listing 8.13: BabyJubJub Point Addition in ZoKrates

```
1  from "ecc/babyjubjubParams" import main as curve_params
2
3  def add(field[2] pt1, field[2] pt2) -> (field[2]):
4
5      field[10] context = curve_params()
6      field a = context[0]
7      field d = context[1]
8
9      field u1 = pt1[0]
10     field v1 = pt1[1]
11     field u2 = pt2[0]
12     field v2 = pt2[1]
13
14     field uOut = (u1*v2 + v1*u2) / (1 + d*u1*u2*v1*v2)
15     field vOut = (v1*v2 - a*u1*u2) / (1 - d*u1*u2*v1*v2)
16
17     return [uOut, vOut]
```

An implementation of BabyJubJub in the ZoKrates language is part of the ZoKrates standard library. For example, Listing 8.13 shows the implementation of point addition for the BabyJubJub curve.

8.3.3 Digital Signatures

Based on the BabyJubJub elliptic curve implementation previously introduced, we provide an implementation of signature verification for a variant of the EdDSA digital signature scheme [191].

8.3.4 Hashing

Hash functions [295] are a fundamental cryptographic primitive commonly used in the context of blockchains, for example in Merkle trees [241] in the context of simplified payment verification (SPV) [258].

By definition, all hash functions share some key properties, e.g., collision and preimage resistance. Some other properties may only be required in some use cases. An example is *pseudorandomness*, the property of a hash function to produce statistically random output, i.e., output that has no discernible structure. For example, pseudorandomness is a crucial requirement in protocols that rely on security proofs in the random oracle model [32, 206].

Besides having different properties, the efficiency of implementations of hash functions in provable abstractions, e.g., rank-1 constraint systems (R1CSs), varies significantly. The same holds for the evaluation of hash functions by the EVM. Hence, developers need to carefully balance proving efficiency and on-chain hashing cost depending on their use case. To support this process, we provide three different hash functions in the ZoKrates standard library to allow developers to choose the right function depending on their requirements. Refer to Chapter 11 for a performance evaluation of the hash functions subsequently introduced.

Hash functions that are efficiently provable in zk-SNARKs are an active area of research. Security of such novel schemes is still under evaluation of the scientific community, but new proposals, e.g., Poseidon [161], are promising candidates for future zk-SNARK-friendly hash functions.

SHA-256

We provide an implementation of the SHA-256 function from the SHA-2 family of secure hash functions [134]. The hash functions of the SHA-2 family are considered to be pseudorandom.

SHA-256 is available in Ethereum as a pre-compiled contract and thus a hash function that is cheap to evaluate in the EVM. It is directly available in Solidity and widely used in the context of blockchain-based applications. However, the implementation in the ZIR is comparatively expensive, as it is defined for binary in- and outputs and heavily relies on bit manipulation.

Pedersen Hashes

The Pedersen hash function is derived from a commitment scheme published by Pedersen [270]. In this scheme, given a group G with two generators $g, h \in G$ for which the discrete logarithm problem is hard, a message m and a random value r, a Pedersen commitment c is defined as:

$$c(m, r) = g^m * h^s \tag{8.4}$$

The commitment function defined in Equation (8.4) can be repurposed to define a hash function for a fixed input length by replacing secret s through a random initialization value r. Given a group G with publicly known randomly chosen generators $g_0, \ldots, g_{n-1}$, the Pedersen hash function h for a message m with bits $m_0, \ldots, m_{n-1}$ is defined as:

$$h(m_0, \ldots, m_{n-1}) = m_0 * \ldots * g_{n-1}^{m_{n-1}} = \prod_{i=0}^{n-1} g_i^{m_i}$$

This hash function's security is based on the observation that finding a collision or a preimage for such a hash would require solving the discrete logarithm problem. Pedersen hashes cannot

be assumed to be pseudorandom and should therefore not be used where a hash function serves as a random oracle [225].

For the Pedersen hash function to be efficiently implementable in the ZIR, a group G with a cheap group operation needs to exist. With the BabyJubJub elliptic curve, we introduced such a group in Section 8.3.2. Hopwood [177] specifies a variant of the Pedersen hash function optimized to reduce the number of required constraints. The ZoKrates standard library implementation follows this approach and adapts it for the BabyJubJub curve.

In the EVM, however, operations on the BabyJubJub curve are not natively supported. Therefore, Pedersen hashes are expensive to evaluate on-chain and should be avoided.

By definition, the Pedersen hash function has a fixed-length binary input and outputs a group element, i.e., a point on the BabyJubJub elliptic curve in our case. Thus, if a hash value is supposed to be used as input again — in Merkle trees, for example — an intermediary binary decomposition step of that hash value is required and has to be accounted for.

MiMC

The MiMC hash function was designed by using the MiMC-Feistel permutation [6] over a prime field in a sponge construction [45] to arrive at a secure and — at the same time — efficiently provable hash function.

The construction is based on established hash function design principles from symmetric cryptography but is still novel and under review by the cryptography community. In this context, there exists a bounty program that incentivizes finding collisions [129]. MiMC hashes are considered to be pseudorandom.

Due to its native use of prime field arithmetics, MiMC hash functions are efficient in circuits. At the same time, they can be evaluated by the EVM with comparatively little overhead.

The MiMC hash function maps from field elements to field elements; applying the function to its output again does not introduce overhead. Packing a binary input into a field element so that the hash function can be applied requires only one constraint (see Section 7.3.2).

8.4 Compilation

In this section, we describe the translation process of a program written in the ZoKrates programming language into the ZoKrates intermediate representation (ZIR). As explained in Section 7.4, this intermediate representation serves as the basis for program execution and generation of proofs attesting execution correctness.

We describe the translation process holistically, focussing on the steps and stages involved in converting a ZoKrates program to the ZoKrates intermediate representation. An overview of this process is shown in Figure 8.1. It depicts the steps in the compilation process as well as the different intermediate representations involved in it.

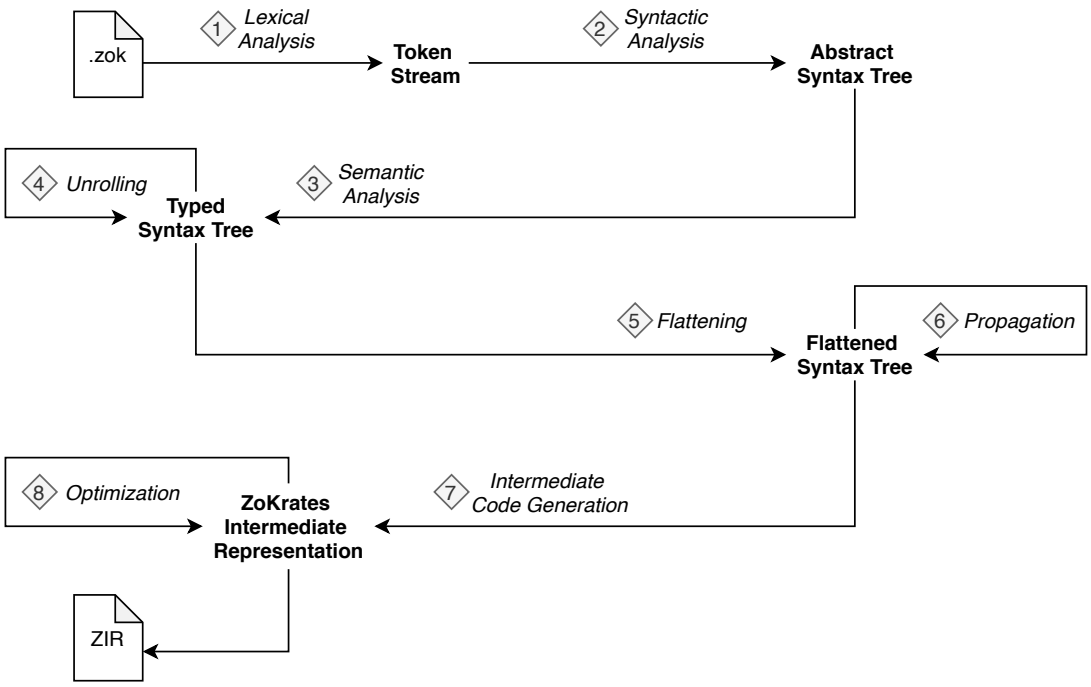


Figure 8.1: Frontend-part of ZoKrates Compilation Process: `.zok` to ZIR.

Herein, the tokenization, syntactic analysis, and semantic analysis steps follow the standard approach for modern compilers [4]. The steps thereafter are specific to our problem domain and cannot directly be mapped to standard steps in compiler design. Still, established standard techniques are applied where possible, albeit with a different goal in mind.

A program in the ZIR format can directly be interpreted for execution, but proving execution correctness involves different backends, i.e., implementations of proving schemes (see Chapter 9), that have different internal constraint representations and data structures.

Such backend-specific representations are generated in the constraint generation process, which involves the constraint generation itself and an optimization step, as depicted in Figure 8.2.

This section does not treat the translation of specific syntactic elements. We explained the translation of the ZoKrates language elements in Section 8.2. The description of the compilation of programs written in the ZoKrates language is purely process-oriented; we map these steps to components of the ZoKrates architecture in Chapter 9.

8.4.1 Lexical Analysis

In the *lexical analysis* step, the ZoKrates source code is converted to a token stream ①, whereby whitespaces and comments are ignored.

This stream of tokens consists of valid ZoKrates identifiers, literals, and keywords, as defined in Section 8.2.1. The set of valid tokens is encoded as the set of terminal symbols in the formal

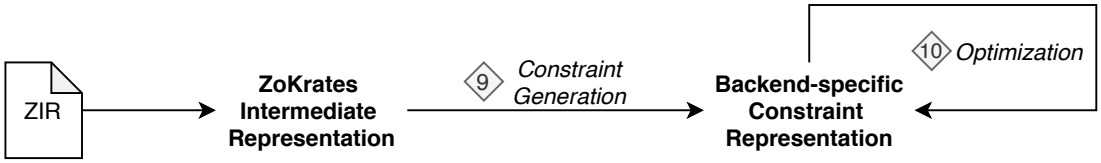


Figure 8.2: ZoKrates Constraint Generation Process: ZIR to Backend-specific Representation

ZoKrates grammar as specified in Appendix A.

In case any undefined tokens are encountered, the lexical analysis step aborts and marks the token's position to aid correction.

8.4.2 Syntactic Analysis

Based on the resulting token stream and the formal grammar's production rules, a tree structure that represents a given ZoKrates program is constructed in the *syntactic analysis* step ②.

This tree representation captures the logical program structure but not every syntactic detail – a fact that reflects in the data structure's name: It is referred to as *abstract syntax tree (AST)* [4].

Unlike in concrete syntax trees, or parse trees, which directly represent the syntactic structure of the source code for a given grammar, concrete syntactic elements, like parentheses or the header of loop-statements, are abstracted from in abstract syntax trees. Only the information relevant in subsequent analysis and processing steps is collected.

In case syntactic analysis fails for a program since it cannot be derived from the ZoKrates grammar rules, a meaningful error message is provided.

In ZoKrates, imports of modules (see Section 8.2.5) are resolved during the syntactic analysis step. Thus, the generated abstract syntax tree always mirrors a program in its entirety.

8.4.3 Semantic Analysis

Semantic analysis ③ ensures that a given ZoKrates program is not only syntactically valid but also semantically well-defined and aligned with the language's specification as given in Section 8.2.

The result of the semantic analysis step is a *typed (abstract) syntax tree* – an AST that is enriched with type information.

Not every program that is syntactically valid is actually sensible. Some rules are inherently hard to encode in formal grammars and are thus checked in a separate semantic analysis step [347].[1]

[1] The ZoKrates grammar is a Parsing Expression Grammar (PEG) (see Appendix A) and thus closely related to context-free grammars [136]. Context-sensitive production rules cannot be modeled using these types of formal grammars.

These semantic rules further restrict the set of valid programs to meaningful and well-defined ones by enforcing additional conditions.

For a ZoKrates program, there is a wide range of semantic rules that are checked in the semantic analysis step. Examples include but are not limited to:

- There exists exactly one main function in a program.
- A variable is always declared before it is used.
- A function always has a return statement, and it is the last statement of its body.
- Logical operators can only be applied to values of type `bool`.

The semantic rules for ZoKrates programs fall into the following two categories besides few exceptions:

1. **Scope Rules:** As described in Section 8.2, the ZoKrates language is statically scoped and allows overloading but no shadowing of variables. These conditions and other rules that depend on where declarations apply — for example, uniqueness of identifiers and validity of variable accesses — belong to this category.
2. **Type Rules:** Type rules are inference rules that define validity and result types of operations on ZoKrates program constructs based on their type as defined in Section 8.2. For example, they ensure that operators are applied to variables of compatible types, and functions are called with parameter types matching the function's signature. The task of determining whether these rules are fulfilled is commonly referred to as type checking. Type information deduced during this process, e.g., the result type of an expression, is added to the typed AST.

Like in previous analysis steps, rule violations result in abortion of the translation process and the generation of a meaningful error message.

The semantic analysis step concludes the analysis phase. The resulting tree-based program representation is transformed into an intermediate representation in subsequent steps.

8.4.4 Unrolling

After the previous standard compilation steps of lexical, syntactic, and semantic analyses, the intermediate representation derivation steps are unlike those found in standard compilers due to the special nature of the ZoKrates intermediate representation.

As previously explained, the ZIR has a purely sequential structure without branches, jumps, or loops. The *unrolling* step ④ is the first of a series of steps that transform the tree structure containing function calls and loops to a sequential instruction list similar to straight-line programs (SLPs) (see Section 7.2.2). This transformation is always possible since the ZoKrates language was explicitly designed with this requirement in mind.

For this purpose, the following two core transformations are performed in the unrolling step:

- **Loop Unrolling:** Since there are no jumps in the ZIR, the target intermediate representation, all loops have to be unrolled to a sequence of statements. Loops are unrolled by replacing the original loop with its loop-body number-of-iterations–times. This transformation, which is explained in detail in Section 8.2.4, requires loop bounds to be compile-time constants. For nested loops, the bounds can depend on outer loops. The compilation aborts with an error message in case of violations of these conditions.
- **Function Inlining:** Like loops, function calls cannot be realized through jumps in the ZIR. Function inlining is an established optimization technique used in compiler design to achieve higher processing speed at the cost of increased program size: A call to a function is resolved by replacing the call instruction with the function's body [4]. In ZoKrates, function inlining is leveraged as a sequentialization technique and applied to all function calls (see Section 8.2.4).

The unrolling step maps the typed syntax tree onto itself and does not introduce a new data structure. Besides the previously introduced core unrolling transformations, there exist further related operations for which the typed syntax tree represents the right level of abstraction.

This includes the following optimization steps related to function inlining:

- **Constant Propagation:** In this standard optimization technique, variables of known constant value are replaced by that constant value [253]. This simplifies expressions throughout ZoKrates programs and reduces the number of constraints in the ZIR. Consider the following example:

```
field a=2
field b=3
field c = a+b // can be replaced by `field c=5`
```

- **Memoization:** During unrolling, function calls that occur multiple times with the same arguments are optimized through *memoization* [4]: The first occurrence of a function call requires an evaluation of the called function for the given arguments. For every other occurrence of that call with the same arguments, the result obtained from the first evaluation can be re-used. The following example shows a sample ZoKrates program which benefits from memoization:

```
def square(field a) -> (field):
  return a ** 2

def main(field a) -> (field):
  return square(a) + square(a) // one evaluation of `square`
```

The last transformation in the unrolling step is neither a sequentialization nor an optimization, but a transformation ensuring correctness by constraining inputs of type `bool`. It is performed in the unrolling step as this step operates on the typed syntax tree abstraction, which contains all necessary information.

As explained in Section 8.2.2, field elements are the only data type in the ZoKrates intermediate representation. Thus, boolean variables are represented as field elements with values 0 or 1 after compilation to the ZIR.

Within ZoKrates programs, the typing rules checked and enforced during semantic analysis ensure that no operations on boolean variables invalidly leave the boolean domain. However, while the type system can ensure the internal validity of ZoKrates programs, its rules rely on a proposition: Only if an input value of type `bool` is actually boolean — and is thus correctly represented as a field element with value 0 or 1 in the corresponding ZIR — correctness is guaranteed.

Since input values are dynamic information that cannot be determined during compilation, we cannot rely on the type system (or any other component of the compiler) to detect field element inputs that are not in $\{0, 1\} \subset \mathbb{F}_p$.

To address this issue, we enforce binarity on input variables by adding assertions to the ZoKrates program that can only be fulfilled for field element inputs in $\{0, 1\}$: For each boolean input variable $x_i \in \{\texttt{true}, \texttt{false}\}$, we demand:

```
assert(x_i == x_i && x_i)
```

During the subsequent flattening step (see Section 8.4.5), this assertion translates to the following ZIR-constraint for field elements $x_i \in \mathbb{F}_p$:

$$x_i == x_i * x_i$$

This equation is only fulfilled for $x_i \in \{0, 1\} \subset \mathbb{F}_p$ and we thus enforced the correct representation of boolean inputs as field elements.

There is one exception with regards to the transformations performed in the unrolling step: ZoKrates programs can import a special type of functions, so-called *embeds*. These functions wrap low-level circuit imports where constraints and witness generation logic are directly imported into ZoKrates. Embeds are used internally in the implementation of the ZoKrates standard library, e.g., to import a hand-optimized SHA-256 compression function. They are introduced in more detail in Chapter 9. Embed functions directly rely on constraint systems internally, and the typed syntax tree used in the unrolling step does not natively capture these concepts. Thus, embeds are an exception to function inlining. We only resolve calls to them during the subsequently introduced flattening step (see Section 8.4.5), as the thereby generated flattened syntax tree provides the appropriate level of abstraction.

8.4.5 Flattening

By definition, the only data type available in the ZoKrates intermediate representation is the field element type, which automatically limits the available operations to field arithmetics, i.e., addition and multiplication (see Section 7.4).

The typed syntax tree, however, represents programs that contain arbitrary ZoKrates data types such as structs, arrays, and booleans. Variables of these types can be linked through arbitrary operators and involve conditionals.

Representing the key step in the transformation of the ZoKrates language to the ZoKrates intermediate representation, the *flattening* step ⟨5⟩ transforms data types, operators, and yet unresolved control flow rules into an equivalent representation consisting of nothing but statements arithmetically linking field elements and directives. Directives are introduced where variable values need to be calculated by a solver (see Section 7.4). The concrete translation idea for each of the syntactic elements of the ZoKrates language, i.e., their arithmetization, was described in detail in Section 8.2.

Flattening transforms the typed syntax tree into a *flattened syntax tree*. This data structure is significantly less complex and already close to the ZoKrates intermediate representation: All variable values are field elements, and a program is represented as a list of statements, where a statement can be a directive, an assignment, an assertion or a return. The flattened abstract syntax tree is in static single-assignment form (SSA) [4], i.e., a variable is assigned exactly once. This structure mirrors the ZIR perfectly, where variables are global and cannot be re-assigned.

Embeds, the special functions that were not inlined during the unrolling step (see Section 8.4.4), are resolved during flattening. Like in the case of regular function calls, this happens through inlining, i.e., by replacing call instructions to an embed with the embed function's body.

8.4.6 Propagation

Several optimizations have already been performed during the previous translation steps, e.g., memoization and constant propagation. However, the arithmetization of ZoKrates language elements in the flattening step introduced new variables and constants again. Furthermore, calls to embeds were resolved.

Hence, we have to expect that there exist known constant values for variables that have not yet been propagated. Thus, in this *propagation* step ⟨6⟩, which maps the flattened syntax tree into itself, another round of constant propagation as described in Section 8.4.4 is performed. Replacement of variables with constant values significantly reduces the number of ZIR constraints, since scalar multiplication — multiplication with a constant — can be encoded in the linear part of an existing constraint instead of requiring a separate one.

8.4.7 Intermediate Code Generation

In the *Intermediate Code Generation* step ⟨7⟩, a program specification in the *ZoKrates intermediate representation* is generated from its flattened syntax tree representation.

The most complex transformations involved in translating a ZoKrates language program to constraints and directives have already been performed in the unrolling and flattening steps. The intermediate code generation step further simplifies the program structure by merging assignments and assertions into constraints as well as removing superfluous tree information. The resulting

well-formed ZIR program complies with the formal ZIR specification provided in Section 7.4. As such, it can be directly executed by the ZoKrates interpreter (see Chapter 9).

8.4.8 ZIR Optimization

The generated ZoKrates intermediate representation may not be optimal. It can include sets of constraints that have more efficient equivalent representations or are not required at all.

Thus, in the *ZIR optimization* step ⑧, such inefficiencies are detected and resolved by applying the following optimizations:

- **Redefinition Removal:** Constraints that represent pure redefinitions of variables can trivially be removed. Consider the following example:

```
// The two constraints
x_2 == x_1
x_3 == x_2 + x_1

// can be replaced by
x_3 == x_1 + x_1
```

- **Tautology Removal:** Constraints that are fulfilled by all possible variable values are obsolete and can be removed.

```
x_1 == x_1 * 1 // is always fulfilled and can be omitted
```

- **Duplicate Removal:** All occurrences but one of a specific constraint can be removed without changing the set of valid solutions for a constraint system.

```
x_3 == x_1 * x_2
x_3 == x_1 * x_2 // is equivalent and can be removed
```

  This idea can be extended to sets of constraints which have the same solutions, i.e., are mathematically equivalent. However, identifying such sets is computationally hard in the general case.

- **Linear Constraint Embedding:** Purely linear constraints, i.e., constraints where the left and the right side are linear terms, can be eliminated by embedding them in quadratic constraints. Consider the following example:

```
// The  following two constraints
x_3 + x_4 == x_1 + 2 * x_2 // linear, scalar multiplication
x_5 == x_3 * x_4 // quadratic

// can be replaced by a single constraint
x_5 == x_3 * (x_1 + 2 * x_2 + (p-1) * x_3)
// through substitution with
// x_4 = x_1 + 2 * x_2 + (p-1) * x_3
```

This idea was initially introduced for rank-1 constraint systems in Section 7.2.3.

After these optimization steps, the ZIR is serialized to a binary format, which concludes the frontend-related part of the compilation process. Optionally, a human-readable representation of the ZIR in the ZoKrates text format (ZTF) can be created in this step (see Section 7.4).

8.4.9 Constraint Generation

In the *constraint generation* step, the constraints contained in the ZIR format are transformed to the target backend's constraint representation to enable proof generation ⟨9⟩. Contained directives are dropped in this process since they are only relevant during interpretation.

This step resembles the code generation step in traditional compilers, where an intermediate representation is converted to the instruction set of a specific target platform. Unlike in that case, however, the ZoKrates intermediate representation is directly interpretable (comparable to Java Bytecode, which is interpreted by the Java virtual machine (JVM)). The constraint generation step is performed solely to support the process of proving execution correctness and is unrelated to the execution itself.

8.4.10 Backend-specific Optimization

In a final step, optimizations that are specific to a backend's constraint representation can be performed if applicable ⟨10⟩.

Since the ZIR is already heavily optimized, such optimizations may not decrease the number of constraints and with that theoretic proving efficiency. However, in practice, the efficiency of proof generation can be improved by using internal data structures efficiently, which decreases memory consumption and processing load.

8.5 Conclusion

In this chapter, we motivated the need for a domain-specific language targeting the previously derived ZoKrates intermediate representation as an abstraction compatible with zk-SNARK proofs.

From domain-specific challenges, we extracted design principles for this DSL. Following these design principles, we specified the ZoKrates language on a syntactic and semantic level. Simultaneously, we described how the syntactic elements can be efficiently translated to the ZIR target abstraction.

Building on this language specification, we introduced the ZoKrates standard library as a collection of useful functions for common tasks in the context of off-chain computations. This extensible standard library is written in the ZoKrates language itself.

In the last section of this chapter, we explained the compilation process from programs written in the ZoKrates language to the ZIR abstraction for execution and backend-specific constraint formats for proving execution correctness.

Together, the ZoKrates language specification and the ZoKrates standard library fulfill the design goals for a programming abstraction that bridges the gap between decentralized application development and cryptographic verifiable computation schemes. The language can express general computations, offers a convenient high-level syntax, and ensures efficiency through purposeful language design and inbuilt optimizations.

In subsequent chapters, we describe the architecture of the ZoKrates framework and its implementation in more detail. The framework's practical viability is evaluated in the context of actual applications in Part IV.

CHAPTER 9

ZoKrates Architecture

In this chapter, we describe the architecture of the ZoKrates framework. An overview of this architecture is provided in Figure 9.1.

The architecture follows naturally from the ZoKrates-based off-chain computation process, as described in Section 6.2. A command-line interface serves as the single entry point and interacts with dedicated components that support tasks like compilation, program execution, or proof-generation, for example.

In the architecture overview (Figure 9.1), we distinguish between two types of inter-component communication: direct communication, where one component directly invokes another component, and indirect communication where information flow is decoupled from control flow and information materializes in files.

The provided architecture is purely conceptual but does serve as a blueprint for the implementation introduced in Chapter 10. While this implementation is specific to the Ethereum blockchain, the architecture introduced in this section is more generic. It allows for arbitrary blockchains that have an execution environment sufficiently powerful for proof verification. Additionally, any implementation of a verifiable computation scheme — a so-called backend — is supported by the architecture's modular design as long as the implemented scheme is compatible with the ZoKrates intermediate representation (ZIR).

Conceptually, the architecture is even compatible with intermediate representations other than the ZIR and could thus support proving schemes that rely on different basic abstractions than rank-1 constraint system (R1CS) or arithmetic circuits. Such a change, however, would require a considerable internal redesign of most components, most prominently the compiler and interpreter.

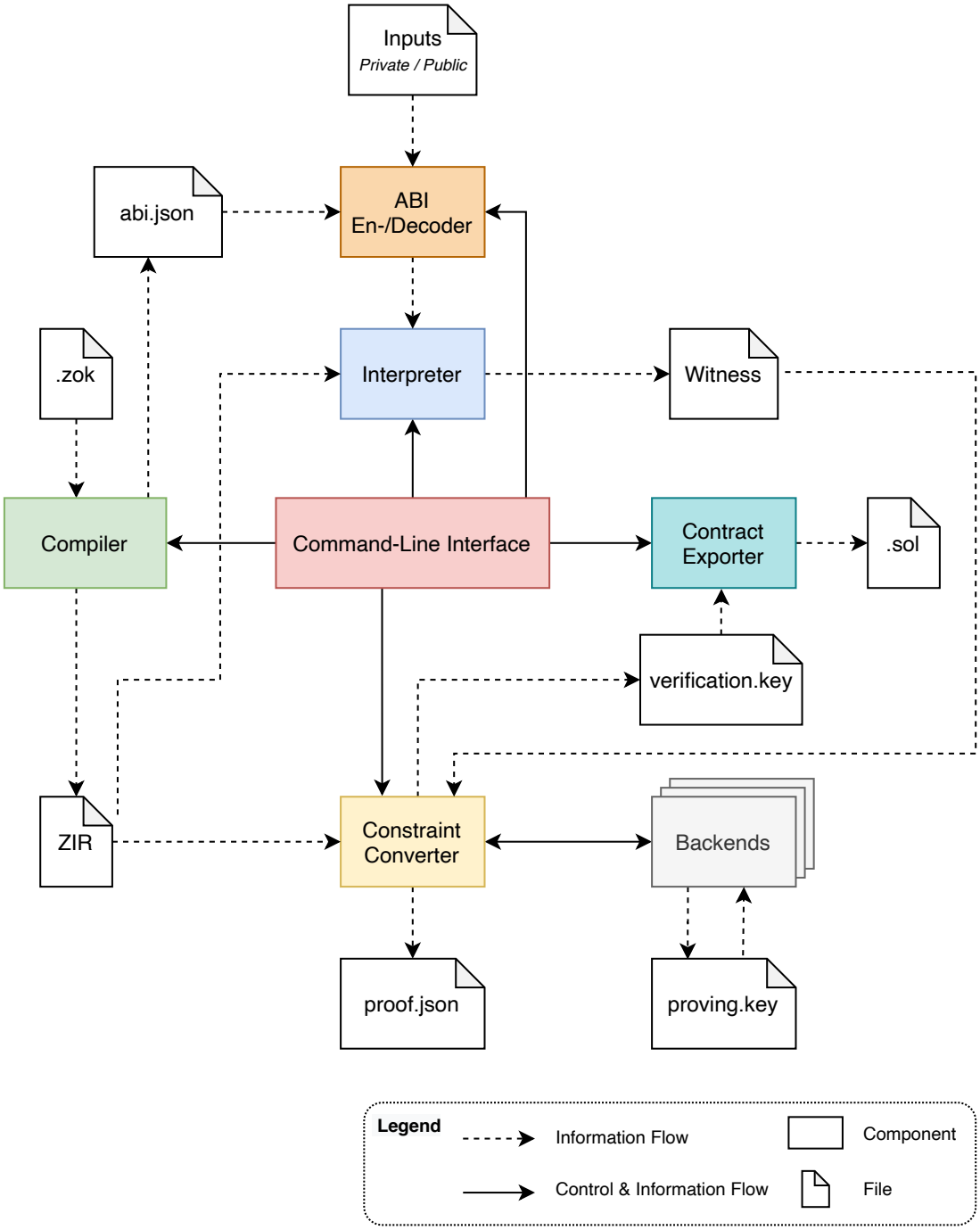


Figure 9.1: Overview of the ZoKrates Architecture

9.1 Command-Line Interface

The *command-line interface (CLI)* is the central piece of the architecture, as it coordinates control flow and orchestrates the interaction between other components. It serves as the user interface for the ZoKrates framework and represents its entry point.

Section 9.1 shows the set of commands exposed by the CLI as an excerpt of the `zokrates help` command's output. These commands directly map to or support the steps in the ZoKrates process described in Section 6.2 from a user-centric perspective.

Since the CLI acts as the central coordinator in the ZoKrates architecture, understanding its role in the context of the different commands provides detailed insight into the relationship between components. Nevertheless, we list but do not explicitly describe the commands offered by the CLI in this section; for clarity, this description is contained in the subsequent sections on the ZoKrates components that are invoked to realize the corresponding commands' functionality.

```
USAGE:
  zokrates <SUBCOMMAND>

FLAGS:
  -h, --help       Prints help information
  -V, --version    Prints version information

SUBCOMMANDS:
  compile          Compiles `.zok` program into ZoKrates Intermediate
      Representation (ZIR). Produces two files: binary ZIR file and human-
      readable '.ztf' file for debugging
  compute-witness  Calculates a witness for a given ZIR program
  export-verifier  Exports a verifier as Solidity smart contract
  generate-proof   Calculates a proof for a given ZIR program and witness.
  help             Prints this message or the help of the given subcommand(s)
  print-proof      Prints proof in chosen format [remix, json]
  setup            Performs a trusted setup for a given ZIR program
```

9.2 Compiler

The *compiler* component is responsible for translating programs written in the ZoKrates language into the ZIR.

The CLI invokes the compiler when receiving the `compile` command. As input, the compiler expects a ZoKrates language file with a `.zok` filename extension that was written by a user. It generates a ZIR file and an optional human-readable program description in the ZoKrates text format (ZTF) for debugging purposes as output.

Internally, the compiler performs a set of translation steps described in detail in Section 8.4. At a high level, it parses the input ZoKrates program according to the grammar provided in Appendix A before sequentializing and arithmetizing the program into a set of constraints and

directives in the ZIR format.

9.3 Interpreter

The *interpreter* is the component that executes compiled programs in the ZIR format and obtains a witness — a type of execution trace — in the process.

The execution of a program through the interpreter is triggered by the CLI on receipt of the `compute-witness` command. Besides the program to be run in the ZIR format, the interpreter receives a set of input values as specified in the original ZoKrates program's `main` function signature. The interpreter does not distinguish between private and public inputs. The compiler encodes this information in the ZIR and the backends account for it.

During interpretation, a witness that satisfies the ZIR constraints is derived and serialized to a file. Recall from Section 7.4 that the ZIR consists of a list of statements where each statement is of one of three types: definition constraints, assertion constraints, or directives. Constraints pose conditions on values of a known set of variables and directives call external solver functions to obtain such values. By definition, the ZIR is in static single-assignment form (SSA) (see Section 8.4.5).

Sequentially traversing the ZIR, the interpreter evaluates definition constraints and directives step by step to find values that fulfill the ZIR-constraints. Since assertion constraints never introduce new values, they do not need to be considered. In the process, a witness, represented by a set of variable-value pairs, is generated; hereby, the SSA guarantees that each variable has exactly one value. During serialization, only values are considered since the order of variables is well defined and can be derived from the ZIR.

An interpreter implementation can choose to additionally evaluate assertion constraints and abort in case of violations. This approach can be useful to provide early feedback to users. Yet, evaluating assertion constraints is purely optional and does not have any correctness implications. A witness' correctness, i.e., constraint satisfaction, is independently checked in the proving process (or at the very latest during proof verification in case a malicious prover provides an invalid proof).

9.4 Application Binary Interface En-/Decoder

The ZoKrates language provides developers with complex types. These types can be used throughout programs as well as for in- and output variables. From a user's perspective, a program's input variables are specified as parameters of the `main` function in the ZoKrates language.

Compiled programs in the ZIR format, however, only support field elements. Consequently, the interpreter expects a list of field element variables as inputs and also returns such a list.

To address this mismatch between the user's ZoKrates language-level view and the ZIR-internal in- and output representation, we introduce a ZIR application binary interface (ABI) description.

This description is then used by the *application binary interface encoder and decoder* component to translate in- and outputs between the ZoKrates language and ZIR abstractions.

The ZIR application binary interface is an interface that enables interaction with programs in the ZIR format. Unlike an application programming interface (API), which provides an interface at the source code level, an ABI provides an interface for compiled programs. This interface is defined through an ABI description, which contains all information required to interact with ZIR programs; the original ZoKrates language program's source code does not need to be available.

The ABI description defines the structure of a program's in- and outputs at the ZoKrates language level. It stores the name, type, and visibility for each in- and output variable. For variables of composite types, i.e., structs, the ABI description additionally contains the breakdown into basic types.

A ZIR ABI description is generated by the compiler during program translation (see Section 8.4). The ABI en- and decoder then uses this description to translate in- and outputs from the user-supplied typed-dependent high-level format to a ZIR-compliant one and vice versa. This translations allows users to provide inputs and receive outputs in a way that is consistent with the original ZoKrates language program without having to be concerned with ZoKrates internal representations and conversions.

The current implementation (see Chapter 10) uses the JavaScript Object Notation (JSON) to encode ABI descriptions as well as in- and outputs supplied to the en- and decoder component; other structured file formats are possible.

ABI encoding is triggered by the CLI on receipt of the `compute-witness` command: Before a ZIR program can be interpreted, the translation of user inputs from the structured representation based on the ABI description — and in turn on the ZoKrates language types — into a ZIR-compliant list of only field elements needs to take place. After successful interpretation, the decoder performs the reverse transformation for a ZIR program's return values and outputs a structured result representation compliant with the ABI description that reflects the output types of the original ZoKrates language program.

Note that usage of the ABI en-/decoder component is optional. Users can also directly supply inputs as a ZIR-compliant list of field elements.

A similar ABI concept is used by the Solidity smart contract programming language [126]. Solidity provides Contract ABIs to enable interaction with compiled contracts deployed to the Ethereum blockchain.

Example: Subsequently, we describe the encoding and decoding process for a simple program in the ZoKrates language and provide examples for the aforementioned artifacts.

Listing 9.1 shows a basic ZoKrates program that takes a point, consisting of two coordinates, as an input and returns a point with inverted coordinates. A user-defined `Point` struct data type is used for in- and output variables.

Listing 9.1: Point Coordinate Inversion ZoKrates Program

```
1  struct Point {
2    field x
3    field y
4  }
5
6  def main(Point p) -> (Point):
7    return Point {
8      x: p.y,
9      y: p.x
10     }
```

The compiler translates this ZoKrates language program into the binary ZIR format. A human-readable representation of the compiled program in the ZoKrates text format is shown in Listing 9.2. The inputs and outputs have been transformed in the process and are now each represented by two variables. Like all variables in the ZIR, they are field elements.

Listing 9.2: Point Coordinate Inversion ZIR in ZTF

```
1  def main(_0, _1) -> (2):
2    (1 * ~one) * (1 * _1) == 1 * ~out_0
3    (1 * ~one) * (1 * _0) == 1 * ~out_1
4    return ~out_0, ~out_1
```

At the same time, the compiler generates the ABI description depicted in Listing 9.3 for the generated ZIR program. This description captures the structure of the `Point` type used in the original ZoKrates language program but defines an interface to the compiled ZIR program in Listing 9.2 as executed by the interpreter.

Listing 9.3: ZIR ABI Description for Point Inversion ZoKrates Program

```
{
  "inputs": [
    {
      "name": "p",
      "public": true,
      "type": "struct",
      "components": [
        {
          "name": "x",
          "type": "field"
        },
        {
          "name": "y",
          "type": "field"
        }
      ]
    }
  ],
  "outputs": [
    {
      "type": "struct",
      "components": [
        {
          "name": "x",
          "type": "field"
        },
        {
          "name": "y",
          "type": "field"
        }
      ]
    }
  ]
}
```

Based on the ABI description shown in Listing 9.3, the ABI en- and decoder can convert structured JSON inputs to a list of field elements as required by the ZIR program in Listing 9.2.

Listing 9.4 shows such a structured input file for the ZIR program where the input is a `Point` with coordinates `x=3` and `y=7`.

Listing 9.4: JSON Inputs for Point Inversion ZIR Program

```
[
    {
        "x": "3",
        "y": "7"
    }
]
```

Execution of the ZIR program through the interpreter results in two field element variables `~out_0` and `~out_1` as shown in Listing 9.2. For the inputs from Listing 9.4, we expect these variables to have values `~out_0=7` and `~out_1=3`.

Table 9.1: Available Verifiable Computation Schemes and Supporting Backends

VC Scheme \ Backend	Bellman [54]	Libsnark [298]
PGHR13 [268]		x
Groth [164]	x	
GM17 [166]		x

Using the ABI description from Listing 9.3, the ABI en- and decoder converts this list of field elements to the JSON output shown in Listing 9.5 that recovers the `Point` type's structure.

Listing 9.5: JSON Outputs of Point Inversion ZIR Program

```
[
    {
        "x": "7",
        "y": "3"
    }
]
```

9.5 Backends

A *backend* is a component that implements a set of verifiable computation schemes that are compatible with the ZIR abstraction.

As analyzed in Chapter 5, verifiable computation schemes have distinct properties and make different tradeoffs. Such schemes are an active area of cryptography research and we expect new and improved schemes to become available over time.

To account for this rapidly changing environment, the ZoKrates framework takes a modular approach and supports arbitrary verifiable computation schemes through multiple backends that complement or substitute each other. The only requirement is that these schemes are compatible with the ZIR abstraction, i.e., are defined for R1CS, arithmetic circuits, or equivalent abstractions (see Chapter 2).

For a verifiable computation scheme to be available in ZoKrates, it needs to be supported by at least one backend. The ZoKrates implementation introduced in Chapter 10 supports two specific backends, and together these backends support three verifiable computations schemes, as shown in Table 9.1.

Backends can be written in arbitrary programming languages and use custom internal data formats and representations. Therefore, the CLI does not directly invoke specific backends but relies on the constraint converter component, which we introduce in more detail in the next section, for

translation. Subsequently, we describe the operations implemented by all backends and indirectly invoked by the CLI. The exact combination of verifiable computation scheme and backend needs to be supplied to the constraint converter as part of this invocation.

Preprocessing verifiable computation schemes require an initial setup step to be performed as part of the protocol (see Chapter 2). Thus, backends supporting such schemes offer a *setup* operation. This operation is indirectly invoked by the CLI through the constraint converter when a `setup` command is registered. It receives the system of constraint of a ZIR program and computes a proving key and a verification key, two public keys required during proving and verification, respectively.

The proving key is only ever used by the backend that generated it. Hence, it is serialized using the backend's internal serialization logic (compare Figure 9.1). In contrast to that, the verification key is used by the contract exporter component (see Section 9.7) and is thus indirectly serialized through the constraint converter component and stored in a text-based backend-independent verification key format.

During development, a purely local setup procedure facilitates testing. While local setups may be sufficient for specific use cases, a distributed setup procedure based on secure multi-party computations (MPCs) is often required to weaken trust assumptions for production deployments depending on the concrete choice of verifiable computation scheme (see Section 5.2.1). The ZoKrates framework is indifferent to the backend-internal realization and accepts both local and distributed setups. Nonetheless, the setup operations implemented by the backends integrated with the ZoKrates implementation introduced in Chapter 10 are purely local. Separate implementations exist to coordinate MPC–based distributed setups [55, 363].

All backends support a *generate proof* operation, which is indirectly invoked by the CLI on receipt of the `generate-proof` command. This operation implements the proving step, which represents a central element of all verifiable computation schemes: As inputs, it expects a representation of the constraint system contained in a ZIR program, a witness for this constraint system, and a proving key (for preprocessing schemes). In case the provided witness is valid, the backend generates a proof confirming constraint satisfaction. This proof is then serialized in a backend-independent way by the constraint converter and stored as a JSON file.

9.6 Constraint Converter

While the previously introduced backends are compatible with the ZoKrates intermediate representation by definition on a conceptual level, they do use different internal abstractions and implementation-specific formats. Furthermore, backends are only ever concerned with the constraints contained in the ZIR and do not consider directives.

To ensure the seamless integration of various backends into the ZoKrates framework, the *constraint converter* component translates from the ZIR to backend-specific constraint formats while removing directives in the process. The constraint converter acts as an adapter [146] that provides a uniform interface for other components and hides the heterogeneous backend interfaces

while exposing their inner functionality. Wrapping the backend's specific interfaces, this uniform interface exposes *setup* and *proof-generation* operations.

To trigger a backend operation, the CLI invokes the respective constraint converter operation and additionally specifies the desired verifiable computations scheme and backend. The constraint converter, in turn, invokes the selected backend after performing necessary input transformations. Finally, the constraint converter translates the backend's response into a backend-independent representation and passes it back to the CLI.

After a successful invocation of a backend's setup operation, the constraint converter serializes a backend-independent representation of the resulting verification key to a file, which is later used by the contract exporter component. Analogously, the constraint converter serializes the proof returned from a successful invocation of a backend's generate-proof operation to a backend-independent JSON–file.

Depending on the context that a proof is supposed to be used in, different representations are suitable. For example, the right representation of proofs to be submitted to a smart contract for verification depends on the tooling used to interact with smart contracts: The Remix IDE for Ethereum smart contract development [125] expects inputs in a different format than the web3.js JavaScript API [128].

To improve usability and to not require ZoKrates users to convert between different representations themselves, the constraint converter offers an additional *print-proof* operation, which outputs proofs in the desired format. Like all other operations, print-proof is invoked by the CLI on behalf of a user.

9.7 Contract Exporter

The *contract exporter* is responsible for the generation of verification smart contracts. These contracts can then be deployed to a blockchain to verify proofs attesting to the correctness of off-chain computations.

Before the result of an off-chain computation can be relied on and used in further processing, its correctness needs to be verified. To support on-chain verification, the ZoKrates framework introduces the contract exporter component, which exports the verification logic for a selected verifiable computation scheme to a smart contract language. The exported smart contract merely serves as a template, and the generated code can be extended and customized to incorporate additional application-dependent on-chain logic before deployment.

The contract exporter component is invoked by the CLI on receipt of an `export-verifier` command. As input, it requires the verifiable computation scheme and a compatible verification key generated by a backend and converted to a neutral format by the constraint converter during the setup step (unless a verifiable computation scheme without preprocessing is employed).

Using this information, the contract exporter generates a verification smart contract that contains the verification logic for the selected verifiable computation scheme, i.e., implements an on-chain

verifier. The generated contract offers a *verify transaction* operation that receives a proof as well as the public inputs and outputs of the related off-chain computation as inputs. It returns a boolean value indicating verification success and hence the correctness of the off-chain execution. An example for the verify transaction operation was given in Listing 6.2.

Internally, this operation triggers a verify function that executes the scheme-specific verification logic and accepts or rejects the provided proof. We provide an example of this verify function for the Groth16 [164] scheme in Listing 9.6.

Listing 9.6: Solidity Contract Excerpt of Verification Logic for Groth16 [164]

```
1  function verify(uint[] memory input, Proof memory proof) internal returns (
       uint) {
2      uint256 snark_scalar_field = 21888242...5808495617; // truncated for
           readability
3      VerifyingKey memory vk = verifyingKey();
4      require(input.length + 1 == vk.gamma_abc.length);
5      // Compute the linear combination vk_x
6      Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
7      for (uint i = 0; i < input.length; i++) {
8          require(input[i] < snark_scalar_field);
9          vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.gamma_abc[i + 1],
               input[i]));
10     }
11     vk_x = Pairing.addition(vk_x, vk.gamma_abc[0]);
12     if(!Pairing.pairingProd4(
13             proof.a, proof.b,
14             Pairing.negate(vk_x), vk.gamma,
15             Pairing.negate(proof.c), vk.delta,
16             Pairing.negate(vk.a), vk.b)) return 1;
17     return 0;
18 }
```

Conceptually, the smart contract export can occur to any blockchain with a smart contract language that supports the cryptographic operations required by the selected verifiable computation scheme. For the schemes listed in Table 9.1, for example, elliptic curve addition and scalar multiplication, as well as a bilinear pairing operation, need to be supported (see Chapter 2).

The Ethereum blockchain, as selected for the implementation introduced in the next chapter, supports these operations through special pre-compiled contracts, which can be called from Solidity. In the generated Solidity verification contracts, inputs are 256 bit unsigned integers that contain field elements with a maximum size of 254 bit. This encoding was chosen as Solidity does neither support a field element type or an enclosing type with smaller bitwidth.

CHAPTER 10

Implementation

The ZoKrates implementation matured considerably since the publication of our original ZoKrates paper [113] and evolved into an open-source project with multiple contributors. Nevertheless, the core components and ideas have not changed. The ZoKrates implementation is available on GitHub[1] and licensed under the GNU Lesser General Public License v3.0[2].

This implementation is written completely in Rust [203] besides minor pieces of C++ code used to connect the `libsnark` backend [298] and a JavaScript binding. We chose the Rust programming language, as it compiles to very efficient native code, offers strong safety guarantees at compile-time, and provides state-of-the-art build tools and dependency management. Furthermore, it can be directly compiled to WebAssembly [341], which allows ZoKrates to be run in a browser.

After providing an overview of the organization of the codebase, we describe and discuss aspects of the implementation that are insightful to the reader, although they are specific to our solution. In that process, we focus on design choices instead of technicalities.

10.1 Organization of the Codebase

Figure 10.1 provides an overview of the codebase of the ZoKrates implementation. Herein, Rust packages and modules form the basic organizational units. To capture the essence of the codebase's structure instead of technicalities, we made two slight simplifications: Explicit dependencies between packages and modules that are captured through transitive dependencies in the figure are not explicitly shown. Glue modules that exist for purely technical reasons were omitted.

Most architectural components, as described in Chapter 9, are also clearly separated within the implementation. They are either represented as packages or as logical groups that comprise a

[1] `https://github.com/ZoKrates/ZoKrates`

[2] `https://www.gnu.org/licenses/lgpl-3.0.html`

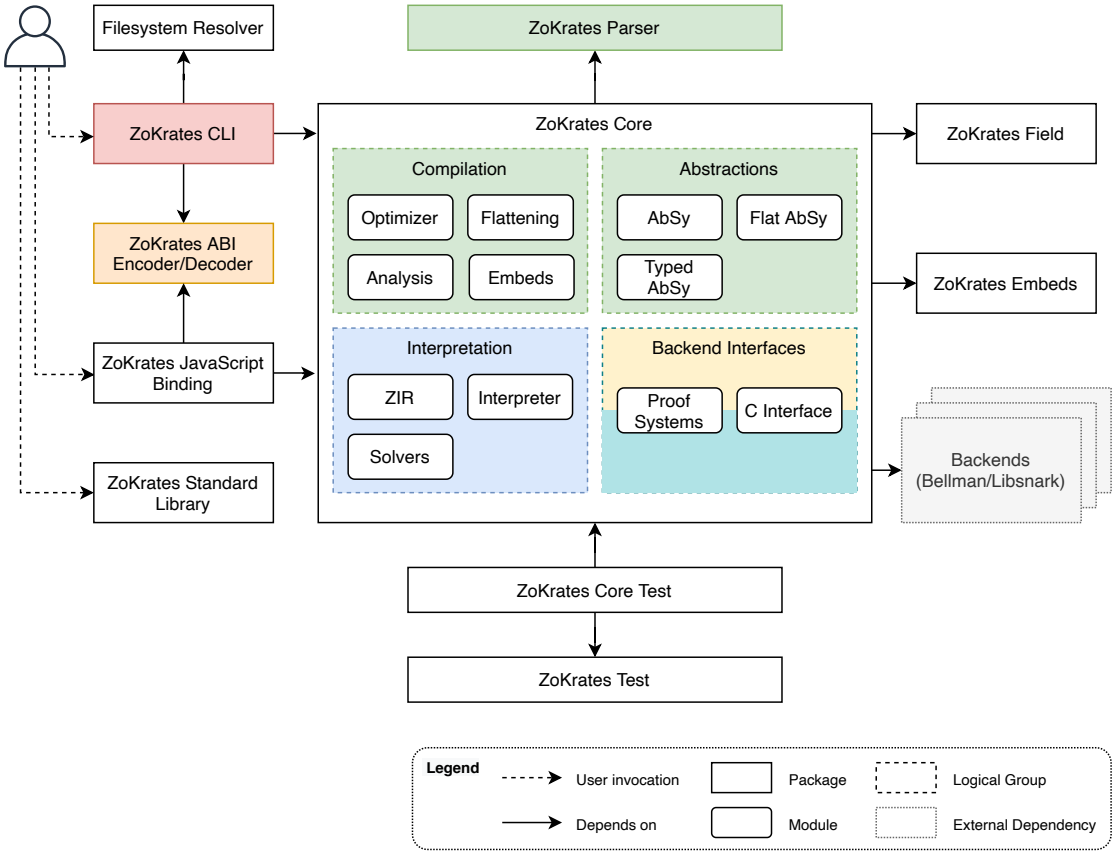


Figure 10.1: Overview of the ZoKrates Codebase

set of related modules. We use the color-coding from Figure 9.1 to indicate which architectural component a package or modules conceptually belongs to.

The ZoKrates CLI is a separate package that depends on ZoKrates core and the application binary interface (ABI) encoder and decoder. The ZoKrates core package is at the heart of the codebase and contains the framework's core functionality. Relying on a set of dependencies, it implements the compiler, interpreter, constraint converter, and contract exporter components of the ZoKrates architecture. The implementations of these components are grouped in one package with well-defined outside interfaces since they are expected not to be used in isolation but as a self-contained unit. Furthermore, the isolation in a separate package allows less critical components, e.g., components that require less rigorous testing, to evolve quicker.

Our implementation differs from the idealized architectural model in some aspects, however. Due to the substantial overlap in required interfaces and data formats, the architectural constraint converter and contract exporter components are implemented through a set of modules in a logical group that we call backend interfaces. This design allows us to avoid duplicate and ensures maintainability. Backends themselves are external dependencies and thus not part of the codebase. The ZoKrates core package only contains backend interfaces. We discuss backend implementations in more detail in Section 10.3.

The codebase contains a set of other packages that do not directly map to architectural components and are thus not color-coded. ZoKrates core can work with arbitrary implementations of finite fields. We provide our field implementation that is specifically designed to be compatible with Ethereum as a separate package to facilitate the integration of other implementations. The core package should not need to be aware of how the information it receives or returns is managed or persisted. How this information is managed depends on the environment that ZoKrates core is executed in. The command-line interface (CLI), for example, directly stores to and reads from the filesystem. When ZoKrates is executed in a browser, in contrast, filesystem access is not available (for more information on browser-based execution, refer to Section 10.6). Thus, filesystem access is not part of ZoKrates core but implemented in a separate filesystem resolver package. The ZoKrates standard library package contains the ZoKrates standard library programs, as described in detail in Chapter 8. The remaining packages are treated in the respective sections of this chapter: We describe our testing and continuous delivery approach in Section 10.8 and discuss embeds in Section 10.4

10.2 Parser

As shown in Figure 10.1, the ZoKrates parser is implemented in a separate package and is therefore not a part, but a dependency of ZoKrates core. The ZoKrates parser package implements the lexical and syntactic analysis processes described in Section 8.4.2, which translate a ZoKrates language program into an abstract syntax tree (AST).

Parsing is a recurring and well-understood problem in compiler design [4]. To support this process, parser generators, which automatically generate parsers from a formal grammar, have been proposed and found widespread adoption, e.g., Bison [105] and ANTLR [269]. In our

implementation, we use pest, an efficient parser generator for Parsing Expression Grammars (PEGs) [136] written in Rust.[3] Pest automatically generates a parser implementation from the formal ZoKrates grammar specification (provided in Appendix A). Replacing an initial hand-written parser implementation, using a parser generator enables faster iterations on the ZoKrates language syntax. Furthermore, grammar-based parser generation helps to avoid inconsistencies between the formal grammar specification and the parser implementation.

Figure 10.2 provides an overview of the components and steps involved in the parsing process in our implementation.

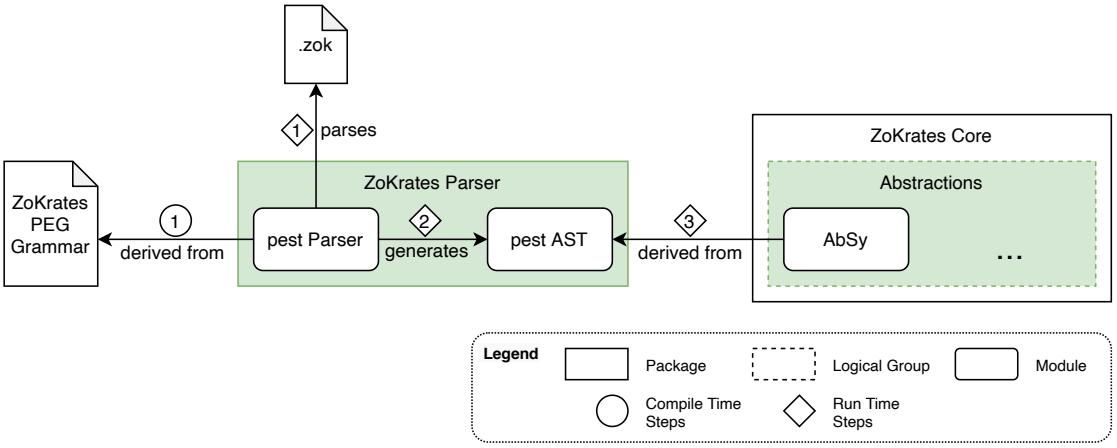


Figure 10.2: Overview of Parsing Implementation

During the compilation of the ZoKrates framework itself, a pest parser instance is derived from the formal ZoKrates Parsing Expression Grammar ①. At runtime, this previously generated pest parser instance parses ZoKrates language programs ◇1 and generates a pest-specific AST ◇2. Then, in an intermediate translation step, the ZoKrates core AST, as implemented in the *absy* module, is derived from the pest-specific AST ◇3. This translation ensures that the ZoKrates core implementation does not depend on a specific parser implementation or parser generator choice. The ZoKrates parser package can be replaced without affecting the abstractions in the ZoKrates core package.

10.3 Backends

In the ZoKrates framework, backends are the components that implement verifiable computation schemes. To ensure security and extensibility, we do not implement verifiable computation schemes ourselves but rely on specialized and dedicated libraries. Thus, backends represent external dependencies of ZoKrates core, as shown in Figure 10.1. As mentioned in Section 9.5, the ZoKrates implementation supports two different backends, Bellman and Libsnark.

Bellman [54] is a Rust library that implements the Groth16 [164] zk-SNARK scheme and was

[3] The official pest website is available at: `https://pest.rs/`

originally developed by Zcash [361]. Since the original codebase only supports the BLS12-381 elliptic curve used in Zcash, we rely on a fork of the original Bellman codebase that incorporates the Ethereum-compatible BN254 elliptic curve.[4]

Libsnark [298] is a library that supports a range of zk-SNARK schemes and is developed by the SCIPR-Lab.[5] The library is written in C++ [186] and is thus — unlike Bellman — not directly compatible with the ZoKrates Rust codebase. Thus, to bridge this gap, the C interface module in the ZoKrates core package implements a compatible foreign function interface, as shown in Figure 10.1.

10.4 Embeds

As explained in Section 8.4.4, ZoKrates introduces embeds as a means to import ZoKrates intermediate representation (ZIR)-compatible circuits implemented outside of ZoKrates and embed them in ZoKrates language functions. For example, embeds can be used to import constraint systems and witness derivation logic from low-level gadget libraries, as provided by Libsnark and Bellman.

The embeds module provides a uniform interface to other ZoKrates components and hides implementation details of imported constraint systems. Internally, an embed implementation contains the imported constraint system and introduces directives and solvers required for witness derivation. As a result, embeds can be imported and used like regular functions in the ZoKrates DSL.

To ensure maximum efficiency, we import a hand-optimized implementation of the SHA-256 compression function from the Zcash sapling implementation [362] as an embed, for example.

10.5 Directives & Solvers

Directives were introduced in Section 7.3.2 as a mean to retrieve additional witness information from the outside world during program execution. Directive statements are contained in ZIR programs to trigger external witness derivation. During program execution, the interpreter resolves these directive statements by calling the associated solver functions that calculate the desired witness information.

The ZoKrates implementation contains the following solver functions:

- `Bits`: As described in Section 7.3.2, the decomposition of field elements to their binary representation requires external witness derivation. This solver supports this process: It accepts a field element i and returns a list of field elements with values in $\{0, 1\}$ that represents the binary encoding of i.

[4] Bellman Community Edition fork available at: `https://github.com/matter-labs/bellman`

[5] The SCIPR-Lab is a cross-institutional academic research collaboration. Its website is available at: `http://www.scipr-lab.org/`

- `Div`: As explained in Section 7.3.2, division cannot be computed efficiently in the ZIR. Thus, we rely on external witness derivation. This solver function accepts two field element inputs, dividend and divisor, and returns a field element that represents the quotient.
- `Sha256Round`: As explained in Section 10.5, the ZoKrates implementation uses an embed to import a hand-optimized version of the SHA256 compression function. The Witness derivation logic for this compression function's constraint system is wrapped in this solver.
- `ConditionEq`: This solver is used as a helper in the flattening of conditionals as described in Section 8.2.3. It accepts one input i and returns outputs o_0, o_1 with:

$$o_0, o_1 = \begin{cases} 0, 1 & i = 0 \\ 1, x^{-1} & i \neq 0 \end{cases}$$

These solvers are required to support the execution of ZIR programs generated by the ZoKrates compiler. The set of solvers can be easily extended in our implementation; they are simple Rust functions.

10.6 WebAssembly and ZoKrates JavaScript Binding

WebAssembly (Wasm) is a virtual instruction set architecture designed to be efficiently implementable and executable in Web browsers. Its goal is to serve as a portable, fast, and safe low-level code format [341]. Native Wasm execution is supported by all major browsers, e.g., Firefox, Chrome, Safari, and Edge.

Since our ZoKrates implementation is written in Rust, it can be compiled to Wasm and run entirely in a browser.

The ZoKrates JavaScript binding facilitates convenient high-level access to the ZoKrates framework from JavaScript code. For that, it exposes ZoKrates' core operations, i.e., the same set of operations exposed by the CLI (see Section 9.1), as JavaScript functions.

10.7 ZoKrates Remix Plugin

Remix is a Web-based integrated development environment (IDE) for Ethereum smart contract development in Solidity. It offers syntax highlighting as well as support for common development tasks such as testing, debugging and contract deployments [125]. Remix can be extended through its plugin engine. This engine enables the integration of other languages and tools into the IDE.

To simplify the development and execution of ZoKrates language programs, Darko Macesic and Edi Sinovic independently developed a ZoKrates Remix plugin.[6] It uses the ZoKrates JavaScript

[6] The open-source implementation of the ZoKrates Remix plugin is available at: `https://github.com/blockchain-it-hr/zokrates-remix-plugin`

binding to interact with a Wasm-based in-browser ZoKrates instance.

The ZoKrates Remix plugin supports all steps of the ZoKrates process (see Section 6.2) via a graphical user interface (GUI). Furthermore, it adds syntax-highlighting for ZoKrates language programs and leverages the ZoKrates ABI for input and output formatting. An example of this GUI is depicted in Figure 10.3.

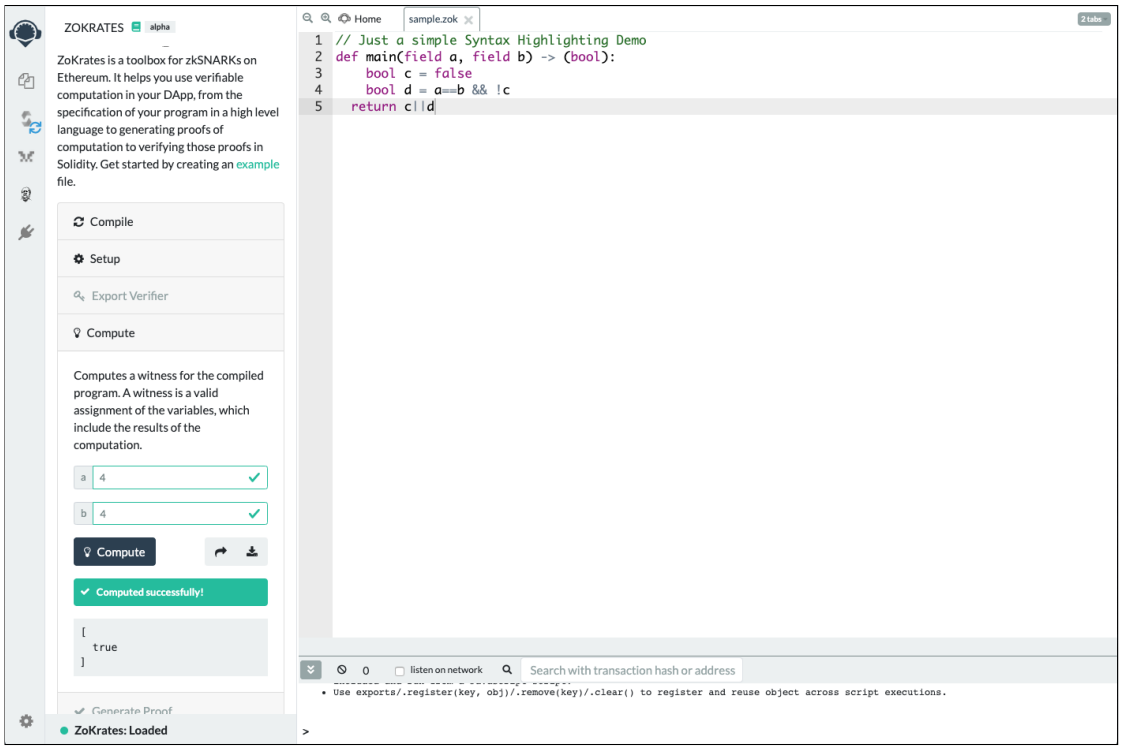


Figure 10.3: ZoKrates Remix Plugin User Interface Example

10.8 Continuous Integration and Delivery

The ZoKrates codebase contains different types of automated tests that are integrated with a continuous integration and delivery (CI/CD) pipeline [28] realized with Circle CI.[7]

We distinguish four different classes of tests within the ZoKrates implementation. These tests comprise larger parts of the framework's functionality in ascending order.

1. **Unit and Integration Tests:** In unit tests, individual modules of the ZoKrates implementation are tested. Integration tests comprise packages that consist of several modules and interact with these through public interfaces. Unit and integration testing are implemented using Rust's testing capabilities.

[7] The Circle CI documentation is available at: `https://circleci.com/docs/2.0/`.

2. **Compilation Tests:** Compilation tests ensure that the compilation of a given piece of ZoKrates language code leads to the expected results or fails in an expected way.
3. **Interpretation Tests:** In contrast to compilation tests, interpretation tests also execute the previously compiled ZoKrates language code. After the successful compilation of a ZoKrates language program, the interpreter runs the resulting ZIR program for a predefined set of inputs, i.e., computes a witness. The result of that computation is then compared to an expected value.
4. **End-to-end Tests:** End-to-end tests comprise all steps of the ZoKrates process: ZoKrates language code is first compiled and then interpreted for given inputs before the result is asserted to equal expected values. Then, after a local setup has been performed successfully, a proof is generated, sent to a local blockchain node, and verified there by the ZoKrates-generated Verification Smart Contract.

The previously described tests are integrated into a continuous integration and delivery process. After a commit has been issued to a branch of the ZoKrates repository, the CI/CD pipeline performs the following steps:

1. **Building:** As a first step, the branch that was updated by the commit is built. For that, a virtual machine is launched, required dependencies are installed, and the respective ZoKrates branch is built from source.
2. **Testing:** On build success, the previously described tests are executed. Furthermore, this step ensures that all source code is properly formatted using a code formatting tool to enforce style guidelines.[8]
3. **Delivery:** The delivery process is only triggered by changes to the master branch of the codebase. This step builds executable artifacts for various target platforms and publishes them to GitHub and DockerHub.[9]

[8] Concretely, we use `rustfmt`, which is available at `https://github.com/rust-lang/rustfmt`.

[9] DockerHub provides a public repository of Docker images. Available at `https://hub.docker.com/`.

CHAPTER 11

Performance Evaluation

We defined three major design goals for ZoKrates in Chapter 6: usability, generality, and efficiency. Of these goals, usability and generality are hard to assess experimentally. Off-chain programs are highly application-specific and need to be carefully crafted for a given context. Efficiency aspects, however, can be captured well in an experimental performance evaluation.

Hence, in this section, we conduct a series of experiments to evaluate the performance of our prototypical implementation introduced in the previous chapter. These experiments serve two goals:

1. They provide an intuition for the performance characteristics of the implementation, especially the relative cost of the supported operations, potential bottlenecks, and starting-points for optimizations.
2. They demonstrate the practicality of the proposed system and its implementation. With that, we address concerns often surrounding approaches that leverage complex cryptographic protocols like verifiable computation schemes.

Experiment Design: Evaluating the performance of an implementation of the ZoKrates framework with its many components and operations is not trivial. In line with our previously defined goals, we focus our experiments on the operations a user invokes when using ZoKrates for off-chain computations and on-chain verification, as described in Chapter 6. Experiments targeting the performance of isolated ZoKrates components are thus omitted.

Of the operations invocable by users, `export-verifier` and `print-proof` are computationally very cheap and can thus be neglected in our performance analysis.

Table 11.1 shows the set of remaining operations for which we subsequently conduct an experimental performance analysis. We categorize these operations into two dimensions:

1. An operation can be an on-chain or an off-chain step.
2. For a given program, it may need to be applied once or repeatedly.

Table 11.1: Categorization of Evaluated ZoKrates Operations

	Off-Chain	On-Chain
Repeated Steps	`compute-witness`, `generate-proof`	`verify`
One-time Steps	`compile`, `setup`	`deploy`

The remainder of this chapter is structured along these dimensions.

Strictly speaking, the on-chain steps are not directly supported by the ZoKrates framework, which only generates the underlying verification smart contract. Nevertheless, these operation's performance characteristics are an important aspect of the overall system's performance evaluation since verification is a crucial step in the off-chain computation process.

Note that the results for repeated steps and one-time steps have to be interpreted from different perspectives: The repeated steps determine off-chain computation and verification performance. Thus, these steps are at the center of attention when interpreting experiment results. Still, the one-time steps provide essential information with regards to overall feasibility: If any of these steps requires more memory than available on given hardware, the ZoKrates system cannot be bootstrapped. Similarly, the smart contract deployment cost must not exceed what can be supported in a block.

Experiment Setup: We now detail the hardware and software that was used to perform the experiments described in subsequent sections. To support independent reproduction of the experiments described in this chapter, we provide fully automated scripts and instructions.[1]

The experiments in this chapter were performed with ZoKrates in version 0.5.2.[2] As a verifiable computation scheme, we selected Groth16 [164] as it represents the current state of the art. The Barreto-Naehrig elliptic curve (BN254) [27] was used to ensure compatibility with Ethereum. We activated the Bellman backend, which supports the former scheme and curve choices (see Chapter 9).

For our experiments, we used standard server hardware with an Intel Xeon E3-1270 v6 CPU@ 3.80 GHz, 64 GiB memory, and an SSD. The Intel Turbo Boost feature was disabled to eliminate variance in the processors' operating frequency and with that ensure consistency in performance over time.

Experiment-specific setup information is provided in the respective section.

[1] Experiment scripts and instructions available at `https://github.com/JacobEberhardt/ZoKratesBenchmarks`.

[2] Executables and Source Code available at `https://github.com/Zokrates/ZoKrates/releases/tag/0.5.2`.

11.1 Off-chain Steps

In this section, we measure the performance of the off-chain steps in the ZoKrates verifiable off-chain computation process. These steps can be executed by arbitrary nodes responsible for program execution and proving independent of the blockchain and directly map to operations offered by ZoKrates.

We first evaluate `witness-computation` and `generate-proof`. These repeated operations are the key determinants of performance since they are performed for every single off-chain computation. Afterward, we evaluate the `compile` and `setup` operations, which need to be performed only once for a ZoKrates program.

We show that the main performance bottleneck for off-chain computations are steps implemented in backends (see Chapter 9) and that compiler optimizations are essential to delivering high performance.

Program Selection: There exists no established set of standard programs for the comparison of off-chain computations yet, and due to the generality of the approach, we expect off-chain programs to be highly application-specific. Thus, we selected ZoKrates programs from the applications introduced in Part IV as a representative, realistic, and relevant set of off-chain programs for our performance evaluation. Refer to the corresponding chapters for a detailed description of these programs.

During the design and implementation of off-chain programs for the applications introduced in Part IV, we identified hash functions as a frequently recurring primitive. They are used to implement aggregators, e.g., Merkle Trees, and to realize commitment schemes. To allow a better insight into how these basic primitives compare, we also include the set of hash functions introduced in Section 8.3 to our list of programs to be measured. The selected programs map two field element inputs to one field element output using a suitable representation for the given hash function.

Experiment Plan: We collect two metrics to characterize the performance of off-chain operations: execution time and memory consumption during execution.

To obtain measurements for the execution time, we utilize the `timeit` Python library.[3] We use the `ps` unix utility to measure memory consumption.

For our experiment results to be meaningful, we need to isolate the experiment runs from external influences as much as possible. Furthermore, we need to account for potential variance between experiment runs.

To satisfy these requirements, we perform experiments repeatedly according to the following experiment protocol:[4]

[3] Timeit Library available at `https://docs.python.org/3.7/library/timeit.html`.

[4] Our experiment protocol is inspired by Tim Peter's thoughts on how to time algorithms [273].

1. Each operation is performed five times and for each repetition, a set of previously defined metrics is collected. Then, for each metric, the average of the five repetitions is calculated. Formally, for n operation loops and the metrics execution time t and memory consumption m:
$$(\bar{t}, \overline{m}) = \left(\frac{1}{n} * \sum_{i=0}^{n-1} t_i,\ \frac{1}{n} * \sum_{i=0}^{n-1} m_i \right)$$

2. Three sets of such repetitions are performed in succession to eliminate any potential impact of parallel processes running on the hardware the experiment is conducted on. As the overall experiment result, the best case repetition with regards to the collected metrics is returned and the standard deviation between repetitions is reported for each metric. In our case, we select the best case purely on time, which leads to the following formal notation for r loop repetitions and standard deviation σ:
$$((\bar{t}, \overline{m})_{min}, \sigma) = \left(\min_t \{(\bar{t}_0, \overline{m}_0), \ldots, (\bar{t}_{r-1}, \overline{m}_{r-1})\}, \sigma\left((\bar{t}_0, \overline{m}_0), \ldots, (\bar{t}_{r-1}, \overline{m}_{r-1})\right) \right)$$
Herein, the $\min_t$ function selects the tuple with the smallest execution time.

Following this protocol, each operation is executed 15 times for each program.

Across all experiments for off-chain steps described in this chapter, the variance in processing time and memory consumption was negligible; we measured standard deviations σ of less than 1%. Thus, for clarity, we abstain from reporting standard deviation explicitly in Tables 11.2 to 11.4.

Expected Results: From a theoretic perspective, computing a witness for a given program in the ZoKrates intermediate representation (ZIR) has a lower time and space complexity than proving satisfaction using a verifiable computation scheme (see Chapters 2 and 7). We thus expect witness-computation to be both faster and less memory intensive than proof-generation.

Expected results are less clear for the one-time steps, namely compilation and setup: The compilation complexity depends on the particular ZoKrates language program and the optimizations applicable in that context. In contrast, the setup step does not depend on the ZoKrates language program but the number of constraints encoded in the compiled ZIR-program.

11.1.1 Repeated Steps

The `witness-computation` and `generate-proof` operations are the main determinants for the performance of off-chain computations. Both operations are run in sequence to first execute a program and then prove the execution's correctness.

Table 11.2 shows the results of the previously introduced experiment plan applied to witness-computation and proof-generation.

We observe that the overall time required to execute and prove an off-chain computation ranges from seconds to few minutes for our given set of real-world ZoKrates programs. These processing

Table 11.2: Time and Memory Consumption for Witness Computation and Proof Generation Steps

Program	Witness Computation		Proof Generation	
	time [s]	memory [MiB]	time [s]	memory [MiB]
Nightfall				
ft-mint	0.6	62.8	3.6	221.5
ft-transfer	9.5	1530.9	99.6	6014.8
ft-batch-transfer	11.1	1686.2	103.5	6570.5
ft-burn	4.7	750.6	48.9	2944.0
nft-mint	0.7	81.1	4.0	286.7
nft-transfer	4.9	787.9	49.6	3113.6
nft-burn	4.8	768.8	49.1	3009.2
Netting				
verify-netting	1.2	158.2	8.2	582.2
Hashing				
SHA256	0.4	43.8	2.2	147.0
pedersen	0.3	4.0	0.8	14.6
mimc	0.3	4.8	0.5	38.4

times are practical for many use cases, especially considering the block confirmation latency that always applies for blockchain-based processing.

Memory consumption ranges from hundreds of mebibytes (MiB) to few gibibytes (GiB). This memory requirement is easily met by modern server and most consumer hardware. Yet, it restricts applicability on mobile and embedded devices.

With regards to the relative cost of the operations, we observe that the witness computation, which is performed by the ZoKrates interpreter component, is strictly faster than the proof generation that happens in a ZoKrates backend, i.e., a framework-external component that implements a verifiable computation scheme. Similarly, the memory consumption for proof generation always exceeds the memory consumption for witness computation.

Thus, the ZoKrates interpreter does not represent a performance bottleneck, even when combined with a highly optimized state-of-the-art backend. Off-chain computation performance is primarily determined by the proof generation step provided by a backend.

Note that the concrete results obtained for the `generate-proof` operation do depend on the selected verifiable computation scheme. In contrast, the `compute-witness` operation is independent.

11.1.2 One-time Steps

The `compile` and `setup` operations have to be executed once for each ZoKrates program before off-chain computations can be performed.

The ZoKrates compiler component supports the compilation operation. The setup operation is implemented by ZoKrates backends. Albeit a mandatory step, it is expected to be mainly used during development and replaced by a secure multi-party computation (MPC)–based setup implementation for production deployments as explained in Section 9.5. Nevertheless, since high performance is crucial also in the development process, we include the setup step in our performance evaluation.

The results obtained by executing our experiment plan for the `compile` and `setup` operations with regards to processing time and memory consumption are displayed in Table 11.2.

The requirements for the compilation and setup steps to be considered practical are weaker than in the case of one-time steps for two reasons:

1. Obviously, they do need to be performed only once for a ZoKrates program, so a higher cost is tolerable.

2. These one-time steps can be performed on different, potentially even dedicated hardware. They do not need to be run on the same node that later performs off-chain computations.

The compilation step is completed within seconds, while the setup step requires considerably more time and takes up to several minutes to process for some programs. Still, for a one-time step, this processing time is certainly acceptable. For both steps, the memory consumptions stay within few gibibytes (GiB).

Table 11.3: Time and Memory Consumption for Compilation and Setup Steps

Program	Compilation		Setup	
	time [s]	memory [MiB]	time [s]	memory [MiB]
Nightfall				
ft-mint	1.1	14.6	31.3	123.5
ft-transfer	23.1	49.9	911.4	3057.1
ft-batch-transfer	26.0	12.0	961.5	3306.5
ft-burn	11.2	4.0	445.8	1528.4
nft-mint	1.4	44.5	38.2	144.8
nft-transfer	11.7	3840.3	459.2	1540.0
nft-burn	11.5	13.3	452.0	1546.7
Netting				
verify-netting	2.5	3306.5	79.1	308.1
Hashing				
sha-256	0.8	1686.2	19.2	76.8
pedersen	1.8	32.5	2.1	12.0
mimc	7.8	29.8	1.0	22.0

In comparison, the compilation step is always significantly faster than the setup step. Also, the `compile` operation generally requires less memory in most cases, but a few exceptions. These special cases are explained through the use of embeds in the corresponding ZoKrates programs (see Section 8.4): Embeds import potentially large external constraint systems and thus require additional memory.

As explained in Chapter 9, the compilation operation is performed by the ZoKrates compiler, whereas a ZoKrates backend is responsible for the setup step. Like in the case of repeated steps, the time required for the backend-supported operation largely determines the overall time spent for one-time steps.

While the measurements for the `compilation` operation apply for any verifiable computation scheme, the results for `setup` do depend on the chosen scheme and the implementing backend.

11.1.3 Compiler Optimization Impact

In this section, we analyze the impact of compiler optimizations on off-chain steps in more detail.

For a given ZoKrates language program, the optimizations performed during compilation directly reduce the number of constraints in the resulting ZoKrates intermediate representation. Since the complexity of all off-chain steps depends on this constraint count, optimizations can have a considerable impact on the overall off-chain computation performance of a program.

With our experiments, we aim to quantify this impact and thereby better understand the optimizations' importance. Furthermore, this analysis provides an insight into the sensitivity of the off-chain steps with regards to constraint count.

To assess the impact of optimizations performed during the compilation of ZoKrates language programs, we modified the ZoKrates compiler so that it does not employ any optimizations beyond what is absolutely needed to compile programs successfully.[5]

In this context, we disabled the following optimization steps (see Section 8.4 for a detailed description):

- All ZIR optimizations
- Constant propagation during unrolling
- Constant propagation after flattening

Table 11.4 shows the results obtained by running our experiment plan for both, the standard ZoKrates version and the derivative with optimizations turned off. Herein, we report values for the constraint count and execution time metrics.

[5] Source Code available at `https://github.com/Zokrates/ZoKrates/tree/benchmarks/unoptimized`.

We observe that the ZoKrates compiler with optimizations enabled produces significantly fewer constraints for all programs. The relation between the optimized and unoptimized constraint count varies between different programs. This result is expected since the specific syntactic choices in the source code determine the potential for compile-time optimizations.

As predicted, the execution times for setup, witness computation, and proof generation are strictly better in the optimized case due to these step's direct dependence on the constraint count. Due to the additional optimization steps, however, the compilation itself takes more time to complete in that case.

From these observations, we conclude that there is a fundamental tradeoff between the processing time of the compilation step and all other steps. This result is explained by the correlation between the other steps' processing time and the number of constraints in the ZIR. However, the experiment results indicate that the performance overhead incurred for optimizations is already overcompensated during the setup step.

Thus, for practical purposes, optimizations are highly beneficial as the setup step already amortizes the additional compilation cost and all other operations directly benefit from the reduced constraint count. Importantly, this includes the repeated steps of witness computation and proof generation, which are the main determinants of off-chain computation performance. Hence, in summary, our experiment results underline the importance of an optimizing compiler in our given context.

The optimizations performed in the ZoKrates framework are independent of specific verifiable computation schemes and the optimization goal — ZIR programs with as few constraints a possible — is shared by all schemes compatible with the ZIR abstraction.

For clarity of the presentation, we do not explicitly display the impact of optimizations on memory consumption, but still report our findings: Like in the case of execution times, there is a strong correlation with constraint count.

Table 11.4: Impact of Compiler Optimizations on Constraint Count and Processing Times

Program	# Constraints		Compilation [s]		Setup [s]		Witness Computation [s]		Proof Generation [s]	
	opt.	unopt.	opt.	unopt.	opt.	unopt.	opt.	unopt.	opt.	unopt.
Nightfall										
ft-mint	86 613	104 411	1.1	0.6	31.3	38.2	0.6	0.6	3.6	3.8
ft-transfer	2 357 944	3 071 248	23.1	9.3	911.4	1188.1	9.5	11.3	99.6	106.8
ft-batch-transfer	2 611 408	3 301 909	26.0	10.4	961.5	1230.2	11.1	12.9	103.5	110.8
ft-burn	1 150 474	1 503 732	11.2	4.7	445.8	585.9	4.7	5.6	48.9	53.4
nft-mint	114 723	133 923	1.4	0.7	38.2	55.2	0.7	0.7	4.0	6.2
nft-transfer	1 208 093	1 567 613	11.7	4.9	459.2	602.4	4.9	5.9	49.6	54.5
nft-burn	1 178 584	1 532 216	11.5	4.8	452.0	593.4	4.8	5.8	49.1	53.8
Netting										
verify-netting	236 601	278 792	2.5	1.2	79.1	115.2	1.2	1.3	8.2	12.1
Hashing										
sha-256	57 238	66 198	0.8	0.5	19.2	27.6	0.4	0.4	2.2	3.1
pedersen	3666	19 180	1.8	1.8	2.1	9.4	0.3	0.3	0.8	2.6
mimc	1321	393 396	7.8	2.6	1.0	196.0	0.3	1.0	0.5	19.9

11.2 On-chain Steps

In this section, we design and conduct experiments to assess the cost of the on-chain steps in the context of ZoKrates-based off-chain computations.

Strictly speaking, the ZoKrates framework does not directly support on-chain proof verification. As described in Chapter 9, the contract exporter component generates a verification smart contract and a proof generated by a backend is provided to users in a standardized format by the constraint converter.

The deployment of this verification smart contract and the submission of the generated proof is the user's responsibility. The on-chain processing costs of these steps directly depend on the ZoKrates-generated verification smart contract. Due to this dependency and to be able to judge the practicality of off-chain computations as a whole, the deployment of the verification smart contract, as well as proof verification through that contract, represent an integral part of our performance evaluation.

Through our experiments, we show that the cost of verifying an off-chain program execution is linear in the number of public inputs and outputs of the corresponding ZoKrates program. Ethereum's block gas limit defines an upper bound for the number of public program inputs and outputs that can be processed.

Metric Selection: Blockchain-based workloads cannot be compared using standard performance metrics such as execution time or memory consumption due to their special execution model: A blockchain's throughput is determined by the block interval and the amount of processing that can occur for each block.

Users pay to purchase shares of this throughput to fulfill their processing requirements. Thus, the core metric for blockchain-based workloads has to be one that captures the size of the share that needs to be acquired to process the workload.

As described in Chapter 2, this idea is captured by the concept of gas in Ethereum. Gas is a metric that combines computational complexity and the use of storage to quantify the cost incurred by a node running a given program in the Ethereum Virtual Machine (EVM). To limit the stress put on nodes, the block gas limit defines an upper bound for the amount of gas that can be consumed in a block. Translating this into our model, the block gas limit represents the amount of processing available in a block.

Thus, in our experiments, we use gas as a metric to capture the consumption of on-chain processing capacity. This approach provides insights into the cost of on-chain processing for users as well as the overall limits of what can be processed in the first place.

Experiment Plan: Recall the following properties of the verification smart contracts:

- The verification smart contract stores a verification key that is specific to an off-chain program. However, this verification key has the same size for all programs.

- The *verify* function receives a fixed-size proof and a fixed number of input variables. These input variables represent public inputs and outputs to the associated ZoKrates program. The gas cost for proof verification depends only on the supported number of inputs.

These observations allow us to choose a more generic experiment design than in the case of off-chain steps, where we used a set of representative ZoKrates language programs from real applications: A single verification contract is sufficient to determine the verification cost for all ZoKrates programs with a constant combined number of public inputs and outputs.

To support our on-chain experiments, we perform the following preparation steps:

- We generate artificial ZoKrates programs with 0 to n public inputs and no outputs, where n is a bounding parameter that can be freely chosen for each experiment.
- We then generate verification contracts with `verify` functions from 0 to n inputs from these programs.
- We execute the previously generated ZoKrates programs for an arbitrary set of valid inputs and generate proofs that attest to the executions' correctness.

Building on these preparation steps, we conduct two experiments to determine the gas cost associated with the on-chain steps.

1. We deploy the generated verification smart contracts to the Ethereum blockchain and determine the gas cost for the deployment in the process.
2. We measure the gas cost for the `verify` function of the previously deployed verification contracts by invoking them with the proofs and inputs generated during the preparation step.

Smart contract execution through the Ethereum Virtual Machine is completely deterministic. This fact greatly simplifies the previously described experiments, since one repetition is sufficient for each operation and input size.

Experiment Setup: In addition to the general hardware and software choices, as described at the beginning of this chapter, additional software components are required for the experimental performance evaluation of on-chain steps.

We use Geth, the Go Ethereum client, version 1.9.13, in a single node deployment to run a local Ethereum blockchain.[6] A local single-node network is sufficient since execution cost for smart contracts in the EVM is independent of the concrete Ethereum deployment. All experiments are based on the Muir Glacier hard fork.[7]

The ZoKrates contract generator outputs a contract in the Solidity programming language. Before it can be executed on the Ethereum blockchain, this contract needs to be translated to EVM

[6] Executables and Source Code available at `https://github.com/ethereum/go-ethereum/releases/tag/v1.9.13`.

[7] Hard fork specified in EIP-2384. Available at `https://eips.ethereum.org/EIPS/eip-2384`.

bytecode. For this translation step, we use solc, the Solidity Compiler, version 0.6.6., with optimizations enabled.[8]

Expected Results: Based on the verification logic of the employed Groth16 verifiable computation scheme, we expect to observe a linear relationship between the gas cost of on-chain verification and the number of inputs supplied to the verification smart contract.

The generated verification smart contracts differ only in the number of expected inputs. Thus, with a growing number of inputs, the size of the verification contract is expected to grow linearly. Since the gas costs for contract deployments linearly depend on the contract's size, we expect an overall linear relationship between the deployment costs and the number of inputs.

Ethereum's block gas limit defines an upper bound for the maximum number of inputs for which on-chain verification can be performed. For on-chain verification to be successful, both contract deployment and verification need to succeed. Thus, whichever operation first exceeds the block gas limit will be the limiting factor for the other one — and with that for the on-chain steps as a whole.

11.2.1 Verification

We first present the results of our verification experiment, as repeated steps generally have a higher significance in the context of our performance evaluation.

For this concrete experiment, we generate a set of trivial ZoKrates programs with 0 to 360 public inputs and no outputs in steps of 20. For the derivation of the upper bound of 360 inputs, refer to the deployment experiment in Section 11.2.2. Subsequently, we run witness-computation and proof-generation for these programs. We export a verification contract for each of the programs and deploy it to the local Ethereum blockchain. After these preparation steps, we run the actual experiment by invoking the verification contracts' `verify` functions with the corresponding proofs and inputs and observing the gas cost.

Figure 11.1 shows the results we obtained from this experiment. Recall that the verification cost for a given number of inputs i is representative for the verification cost of all ZoKrates language programs with a combined number of public inputs and outputs equal to i. For the on-chain verification of a proof and no inputs, $\sim 222\,\mathrm{k}$ gas are required. As can be seen from the graph, the gas costs rise linearly with the $\#inputs$ supplied. For every additional input, the cost increases by $\sim 9.4\,\mathrm{k}$ gas.

To provide a better intuition on the price a user has to pay for on-chain proof verification in fiat currency, we provide a cost estimation in euro for the Ethereum main network in Table 11.5.[9] Note that the estimated prices can vary significantly over time due to their dependence on the volatile ETH/€ exchange rate. To be able to make an estimate, we had to fix a gas price. This

[8] Executables and Source Code available at `https://github.com/ethereum/solidity/releases/tag/v0.6.6`.

[9] The estimations provided in Tables 11.5 and 11.6 were performed using network statistics observed on the Ethereum mainnet on 01.05.2020.

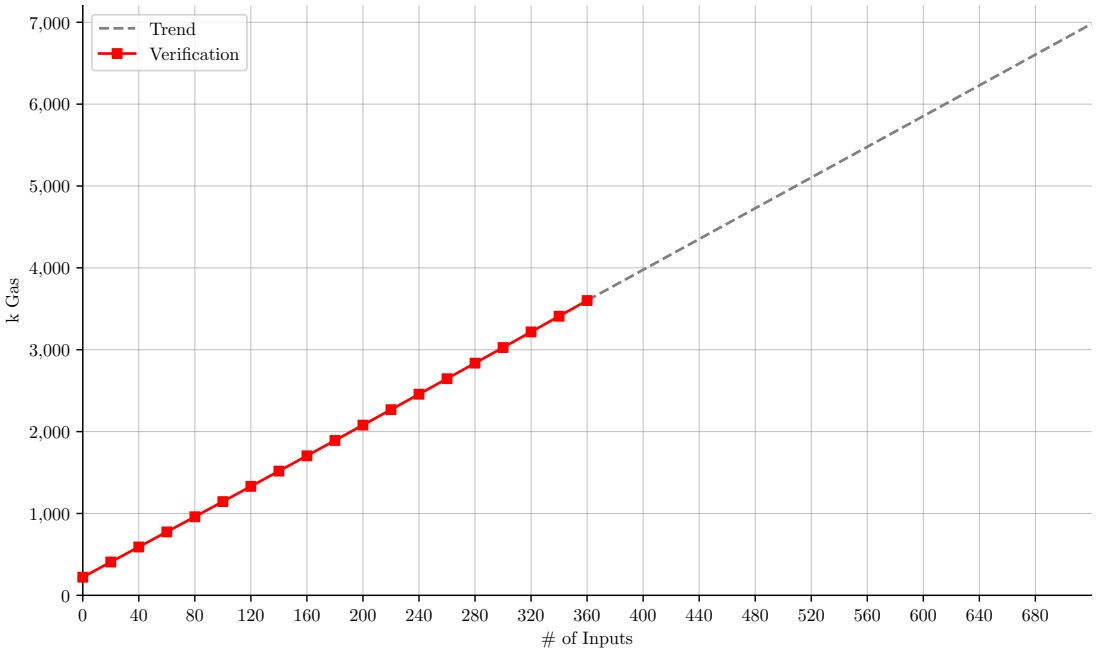


Figure 11.1: On-chain Verification Cost in Gas per # of Inputs

choice significantly impacts the expected latency for and likeliness of on-chain processing. To account for this and demonstrate that our fixed gas price corresponds to a reasonable Quality of Service (QoS), we include a QoS prediction for that price.[10] These estimates are only valid for the Ethereum main network; they do not apply for other deployments. In private Ethereum networks, for example, proof-verification is virtually free.

Besides the monetary cost, the values obtained in our experiment also indicate the maximum verification load that can be processed by a blockchain. The block gas limit defines an upper bound for the number of verification operations per block.

As an instantiation of the delegated computation off-chaining pattern, verifiable off-chain computations can be used as a scalability approach (see Section 5.2.1). Here, the on-chain verification cost determines the break-even point between on- and off-chain processing. Note that this is not a pure cost argument, but simultaneously improves scalability. Computations surpassing the block gas limit can be executed off-chain and then verified on-chain at a fixed cost.

An off-chain computation is beneficial from a scalability and cost perspective if its on-chain verification cost is smaller than the execution cost of the equivalent on-chain computation. Based on our experiment results, this insight yields the following approximate decision rule for when to off-chain a computation:

$$gas_{on-chain} > 221645 + 9387i$$

[10]QoS prediction based on transaction cost calculator available at `https://ethgasstation.info/`.

Table 11.5: On-chain Verification Cost Estimation for Ethereum Mainnet

Metric	Proof Without Inputs	Every Additional Input
Gas Used	221 645	9387
Gas Price (GWei/Gas)	7.8	7.8
ETH price (€/ETH)	190	190
Transaction fee (ETH)	0.00172	0.00007
Transaction fee (€)	0.33	0.01

Estimated Verification Cost: Baseline and Additional Inputs

Metric	Value
% of last 200 blocks accepting this gas price	56.25
Mean Blocks to Confirm	20.9
Mean Time to Confirm (s)	290

Blockchain QoS for Given Gas Price

Here, i represents the combined number of public inputs and outputs of the computation.

This decision rule does not account for privacy benefits, which need a use case dependent evaluation and cannot easily be priced.

11.2.2 Deployment

In this section, we experimentally determine the gas cost of verification contract deployment and discuss the obtained results.

Like in the previous experiment, we generate a set of trivial ZoKrates programs with 0 to 360 public inputs and no outputs in steps of 20. Then, we export a verification contract for each of the programs, which completes the preparation steps. By deploying the previously generated verification contracts to the local Ethereum blockchain, we experimentally determine the deployment gas cost.

In Figure 11.2, we provide an overview of our experiment results. In addition to the cost-of-deployment graph displayed therein, we delineate the current approximate block gas limit that applies for the Ethereum mainnet and amounts to ~10 000 000 gas.[11]

[11]Gas limit statistics obtained from EthStats on 01.05.2020. Available at: `https://ethstats.net/`.

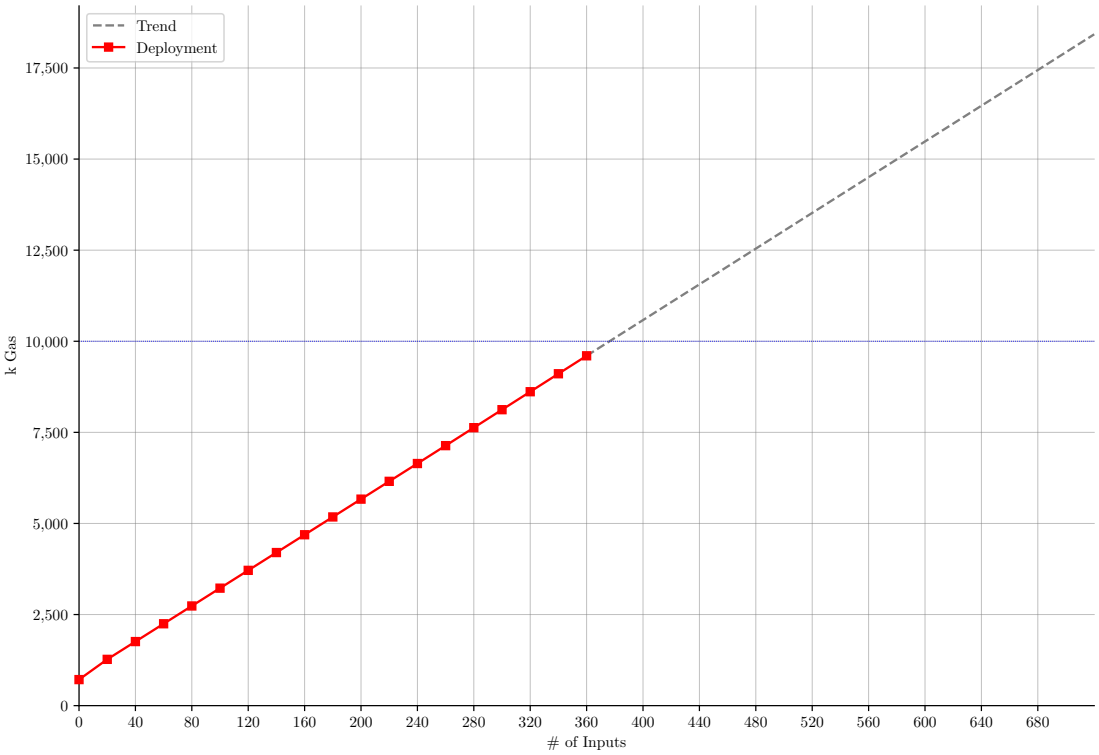


Figure 11.2: Verification Contract Deployment Cost in Gas per # of Inputs

Table 11.6: Verification Smart Contract Deployment Cost Estimation for Ethereum Mainnet

Metric	Contract Without Inputs	Every Additional Input
Gas Used	714 877	24 423
Gas Price (GWei/Gas)	7.8	7.8
ETH price (€/ETH)	190	190
Transaction fee (ETH)	0.00558	0.00019
Transaction fee (€)	1.06	0.04

Estimated Deployment Cost: Baseline and Additional Inputs

Metric	Value
% of last 200 blocks accepting this gas price	56.25
Mean Blocks to Confirm	20.9
Mean Time to Confirm (s)	290

Blockchain QoS for Given Gas Price

We observe that the deployment of a verification contract with no inputs requires $\sim 715\,\mathrm{k}$ gas. Since a contract's size increases with its number of inputs, the deployment cost increases proportionally. More specifically, the deployment cost linearly rises at a rate of $\sim 24.4\,\mathrm{k}$ gas for each additional input to a contract's `verify` function.

Incrementing the number of inputs by another step to 380 would lead to a deployment gas cost that is higher than the block gas limit and thus fail. In a separate experiment, we determined 377 to be the maximum number of inputs for which contract deployment is possible (9 995 302 gas). This input count represents the upper limit for the number of combined public inputs and outputs that a ZoKrates program may not exceed to still be verifiable on the Ethereum mainnet. It is important to note that this limit does not apply to other blockchain deployments. Private Ethereum networks, for example, can — depending on their configuration — support much larger input numbers.

To provide an estimation for the cost that incurs for the deployment of verification contracts in fiat currency, we provide an estimate for the Ethereum mainnet in euro in Table 11.6.

11.3 Conclusion

In this section, we conducted an empirical evaluation of ZoKrates-based off-chain computations. We experimentally assessed the performance and cost of the involved off- and on-chain processing steps and provided a detailed description and interpretation of the individual results. With that, we demonstrated the practicality of our approach and provided an intuition on the performance characteristics of the ZoKrates framework.

Our experiments showed that the performance of off-chain computation steps is largely determined by framework-external backends. The core ZoKrates components do not represent a bottleneck. This result was expected due to the inherent complexity of the implemented verifiable computation schemes. To improve backend performance, graphics processing units (GPUs) or specialized hardware can be leveraged to speed up expensive steps in verifiable computation scheme implementations.[12]. Significant improvements can result from advances in verifiable computation schemes themselves. Due to its modular backend architecture, ZoKrates can directly benefit from independent efforts in this field.

We further showed the importance of optimizing ZoKrates language programs during the compilation step, since these optimizations directly affect the performance of all backend steps. In spite of everything, the most influential performance factor are the ZoKrates language programs themselves. They should be carefully designed for efficiency and leverage primitives optimized for the underlying abstraction, as shown in our experiments for hash functions.

The experiments for the on-chain steps allowed us to identify a limit for the number of public inputs and outputs a ZoKrates language program may have. A verification contract needs to be small enough so that it can be deployed within a target blockchain's block gas limit — a condition violated for a sufficiently large number of inputs.

As a way to overcome this limitation — or more generally to lower the verification cost — we propose the following trick that reduces the number of public inputs to one. Rewrite the ZoKrates program the following way:

- Make all public inputs private.
- Add the calculation of a commitment to these private inputs to the program's body and add the computed commitment as program output.

Modify the verification smart contract generated for this program:

- Extend the `verify` function so that it stores the commitment after successful proof verification.
- Define a `reveal` function that checks a set of inputs against a previously stored commitment.

[12]Results of a GPU-prover implementation competition showed speedups of 3x over a CPU-based reference implementation [261]

An off-chain execution of the ZoKrates program is considered valid if and only if the submitted proof is valid, and the subsequently revealed list of private inputs matches the commitment supplied with the proof. Note that this trick solves the initial problem but comes at the price of a higher program complexity — and with that constraint count — as well as an additional on-chain transaction.

In summary, we showed that ZoKrates-based off-chain computations are practical but also time- and resource-intensive processes. With thoughtful program design, they can be used to address privacy and scalability challenges in blockchain-based applications. Hereby, the ZoKrates framework supports developers with advanced optimizations and a standard library of efficient primitives.

Part IV

Applications

In the previous part, we introduced ZoKrates, the first comprehensive framework for verifiable off-chain computations. Our performance analysis in Chapter 11 showed practicality with regards to performance, but a holistic evaluation needs to also take applicability and practicality into account.

To demonstrate ZoKrates' applicability and practicality, we present applications of ZoKrates-based verifiable off-chain computations in the context of real-world use cases in this part. Concretely, we describe how ZoKrates can be applied to improve the privacy or scalability properties for the three motivational application contexts described in the introduction of this thesis: peer-to-peer energy trading, efficient blockchain interoperability, and token transfers.

In our first application, we use ZoKrates to hide Smart Meter data in future energy networks while still enabling trustless processing for the purpose of sharing energy in a community of households. We implemented and evaluated our solution in the context of BloGPV, a german national research project.

Second, we introduce zkRelay, a scalable blockchain-relay that leverages verifiable off-chain computations to validate for block-header validation. Our ZoKrates-based implementation reduces the cost of validating Bitcoin headers on the Ethereum blockchain up to $187\times$ compared to BTC relay, the state-of-the-art solution.

Nightfall, an open source project independently realized by Ernst & Young Blockchain R&D, uses ZoKrates to create privacy-preserving ERC-20 and ERC-721 token transfers for the public Ethereum network. This independent ZoKrates-based implementation of a complex protocol underlines the maturity of ZoKrates.

Besides these applications, ZoKrates has seen considerable independent use in academia and industry as a privacy and scalability engineering tool [30, 143, 169, 175, 267, 277, 307, 314].

This part is based on and reuses material from our publications at IEEE ICBC 2020 [111] and IEEE EuroS&PW 2020 [343].

CHAPTER 12

Privacy-Preserving Energy Trading

In this chapter, we demonstrate how ZoKrates can be used to enhance the privacy properties of a blockchain-based application in the energy domain.

After motivating the problem and introducing domain-specific background, we introduce a blockchain-based system that leverages zero-knowledge verifiable off-chain computations to facilitate automated energy sharing within a community of households in a trustless and privacy-preserving way. This system protects the participating individual's privacy and at the same time increases the profitability of renewable energy production. We provide a proof-of-concept implementation using the ZoKrates framework and the Ethereum blockchain and conduct an evaluation in the context of BloGPV[1], a major German national research project on blockchain-based energy networks. More generally, we show how ZoKrates-based off-chain computations can be combined with on-chain commitments to execute algorithms in a group of distrusting members with blockchain properties while preserving privacy.

This chapter is based on and contains material from our paper [111] that was published at the IEEE International Conference on Blockchain and Cryptocurrency (IEEE ICBC) 2020.

12.1 Motivation and Overview

Climate change is one of the grand challenges of the century. To prevent further global warming, it is of paramount importance to reduce consumption of fossil energy sources. Increasing production from renewable energy sources is a crucial factor in this transition towards a more sustainable form of energy production. Today, however, economic incentives for the small-scale production of renewable energies are not aligned. The costs of maintaining solar panels, for example, often exceed expected sales profits without government intervention.

A line of work addresses this problem by enabling households to trade energy they produce with each other or on public markets. Several proposals for blockchain-based peer-to-peer energy

[1] `https://blogpv.net/`

trading fall into this category [5, 42, 145, 178, 233, 240, 255].

However, these proposals deliberately do not consider the idiosyncrasies of the energy market and today's physical energy grids to a sufficient degree: Load profiles of individual households are hard to predict, which limits their ability to trade future energy flows [78]. Furthermore, market-based peer-to-peer trading approaches take a greenfield approach and do not consider existing restrictions and peculiarities [222, 233, 247]: Establishing a global, large-scale trading system where arbitrary pairs of households could engage in peer-to-peer trading would be disruptive on a technical and legal level and, therefore, hard to instantiate and deploy.

Therefore, to make an impact today, we propose a localized and community-oriented approach where households in a shared local distribution grid maximize their internal consumption: As of now, individual prosumers maximize the use of their own energy production, i.e., they only purchase the amount of energy from an electricity supplier that they do not produce themselves. In our approach, we expand this idea to a group of households in the same local distribution grid. Together, they form a community that shares the energy produced internally, and the supplier serves as a gateway providing or purchasing the residual load for that community. The locality of prosumption can unlock further economic benefits, e.g., reduced network charges or tax exemptions. Our approach considers the idiosyncrasies of the current energy grid, making it inclusive and realistic to deploy. Neither does it require changes to the physical energy grid, nor household production planning, or a change in participant's behavior.

The instantiation of this approach comes with several technical challenges: A community needs to reliably keep track of its internal production and consumption of energy. From this data, the residual load which needs to be purchased from a supplier to balance the grid can be calculated. We refer to this process of calculating community-internal net production and consumption values as *netting*. For this calculation, households in a community should neither have to trust each other, nor the electricity supplier. However, they can not rely on their calibrated and trusted metering infrastructure either as the netting of energy in a community is not reflected physically. Additionally, energy consumption data is highly sensitive, which further complicates this processing. This data allows detailed insights on household behavior [251] and hence must not be shared within a community.

As our core contribution, we propose a system design to address the challenges previously stated, i.e., we enable the calculation of nettings for a community in a trustless and privacy-preserving way: We leverage ZoKrates to calculate a netting for a community on blockchain-external resources while maintaining the blockchain's trustlessness property. The blockchain verifies this computation's correctness without ever learning the sensitive consumption data, which is hidden through ZoKrates' zero-knowledge property. Furthermore, we provide a characterization of desirable netting results for a community as well as an algorithm to compute such nettings efficiently.

Abstracting from the peculiarities of the specific use case, we provide a more fundamental result: We show how ZoKrates-based off-chain computations can be combined with a commitment scheme to execute a calculation in a group of mutually distrusting members with blockchain properties and privacy-protection at the same time.

To assess the viability of our design, we implemented an open-source prototype[2] of the proposed system for the Ethereum blockchain using the ZoKrates framework. We conducted an extensive evaluation in the context of *BloGPV*, a German national research project on renewable energy production.

12.2 Background

In this section, we provide necessary background on physical energy grids as well as energy markets and a prosumer's economic perspective. Our description is based on the European energy grid [266] but stays sufficiently abstract so that our insights hold for other countries as well.

12.2.1 Energy Grid Organization

The energy grid is a physical network that enables the delivery of energy from producers to consumers. It can most easily be described by the responsibilities of its building blocks:

- **Generation**: Power plants and other producers generate energy they supply to the grid.
- **Transmission**: The transmission grid is responsible for long-distance energy transmission
- **Distribution**: The distribution grid takes care of stepping down energy, so it is usable by consumers and delivery to the location of consumption.
- **Consumptions**: Households and other consumers use the energy provided through the distribution grid.

For the remainder of this chapter, we focus on local distribution grids, which we define as subsets of the distribution grid. Figure 12.1 show an example of a local distribution grid in which a supplier-controlled gateway and several households are directly connected.

12.2.2 Energy Market Organization

The energy market is the virtual place where compensation for utilization of the energy grid is organized. We can identify five logical groups of actors that fulfill distinct roles and participate in this market:

- **Electricity Suppliers**: Enter contracts with prosumers in order to buy and sell energy and balance residual loads in distribution grids.
- **Energy Exchanges**: Trading platforms where utilities trade energy futures.
- **Prosumers**: Produce or consume energy in the grid.
- **Infrastructure System Operators**: Supply and maintain the physical infrastructure for energy transmission. They can be categorized by the layers of the energy grid described

[2] `https://github.com/JacobEberhardt/decentralized-energy-trading`

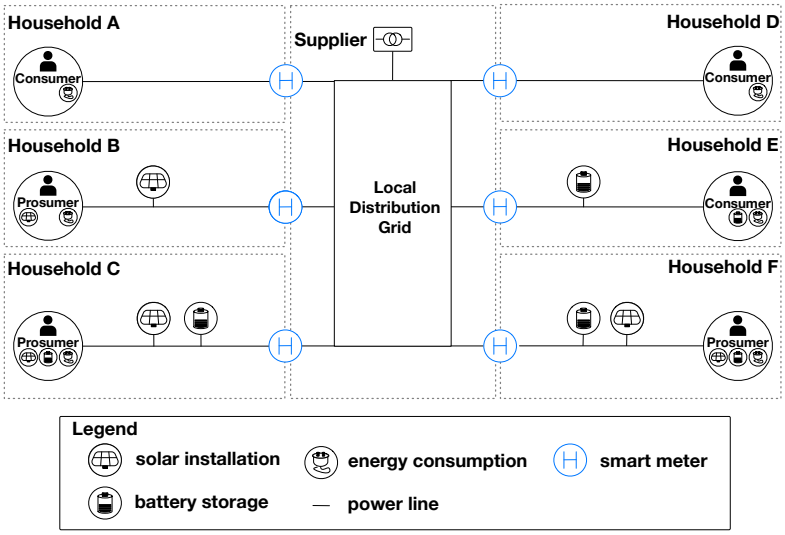


Figure 12.1: Local Distribution Grid formed by Households and Supplier.

in Section 12.2.1, e.g., Transmission System Operators. For their services, they are compensated by the other groups of actors.

- **Meter Operators**: Supply, maintain, and provide access to metering infrastructure and ensure that regulatory requirements are fulfilled. For these services, they are compensated by utilities.

To illustrate the relationship between these actors involved in the energy market, we provide an overview of the communication and compensation flows in Figure 12.2.

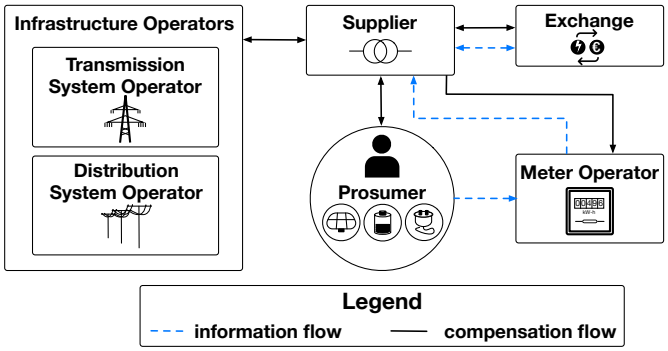


Figure 12.2: Actors and their Relation in the Energy Market

12.2.3 Prosumer Economics

In the current, established model of energy supply, an electricity supplier provides energy to consuming households through a local distribution grid. To ensure that the supplier cannot — intentionally or accidentally — bill the consumers for the incorrect amount of consumed energy,

calibrated electricity meters are installed in households. In the context of this work, these are smart meters, i.e., electronic devices that collect measurements digitally and have communication and processing capabilities. Such a device serves as an independent reference point: Both consumers and suppliers can refer to it in case of disputes with regards to the amount of consumed energy.

For energy consumed, households pay a fixed price p_{buy}, which is contractually agreed on with the supplier in advance. Households that produce energy can sell it to the supplier at price p_{sell}. The supplier sets these prices in a way that it makes a profit. We can express the relation between both prices as $p_{buy} = p_{sell} + \Delta_{margin}$. By netting consumption and production within a community, this margin can be internalized and benefit the participating households [240].

12.3 Related Work

In this section, we present results from literature that are relevant in the context of this chapter and discuss their relation to our contributions. To structure this discussion, we classify related work in three categories: blockchain-based energy trading, privacy in smart grids, and privacy in blockchain-based energy trading.

12.3.1 Blockchain-based Energy Trading

There exists an extensive body of research in the context of blockchain-based energy trading. Numerous proposals have been surveyed by academia [50, 159, 171] and industry [31, 99]. Mengelkamp et al. [240] introduce a blockchain-based local energy market, where households trade future energy flows. Several other proposals suggest a comparable ex-ante auction mechanism to support peer-to-peer energy trading among households [178, 233]. Munsing et al. [255] propose a blockchain-based mechanism for microgrid balancing and control of distributed energy resources. Mihaylov et al. [246, 247] represent renewable energy production as cryptocurrency tokens called NRGcoins. These tokens can be traded and are used in a utility-controlled pricing model to incentivize balancing local grids.

While similar in motivation, these market-based approaches require hard-to-predict future load profiles for individual households and do not consider privacy challenges in their design.

12.3.2 Privacy in Smart Grids

Privacy is recognized as a crucial property in Smart Grids as energy usage data can reveal household behavior and thus represents personally identifiable information [251, 265]. With the evolution of Smart Grids, privacy of consumer data became a key concern within the energy domain and was addressed by research before the advent of peer-to-peer trading [18, 339]. Solutions vary from zero-knowledge proofs [188, 289], hiding & anonymization techniques [115, 194], spatial and temporal aggregation [3, 102], homomorphic encryption, multi-party computations, and differential privacy [2] to simply adding noise.

Unlike our approach, these proposals attempt to hide information from utilities and meter operators. In contrast, we primarily focus on the protection of personally identifiable information from

other peers in a blockchain network. Hence, both lines of work should be combined for optimal privacy protection as they address different concerns.

12.3.3 Privacy in Blockchain-based Energy Trading

More closely related to our approach, there are some efforts to incorporate privacy requirements into blockchain-based designs for energy trading. Dorri et al. [106] propose an off-chain routing method to agree on prices for energy tokens and then execute the actual trade on-chain through an atomic swap which does not reveal participating smart meters. The work is conceptual and there is no implementation and evaluation available. In another token-based approach, Aitzhan et al. [5] propose *PriWatt*, a Bitcoin-based energy trading system that uses Bitmessage for anonymous messaging. While privacy is presented as a property of the proposed system, this does not reflect in the architecture or protocol design. Gai et al. [145] address linking attacks in blockchain-based energy trading by introducing noise in combination with dummy accounts. Laszka et al. [217] propose privacy-preserving energy transactions (PETra), a mixing-based approach to hide the ownership of energy tokens. Bergquist et al. [42] discuss potential improvements of this approach through anonymous routing and advanced mixing techniques.

In summary, these projects attempt to hide on-chain token ownership and transfer history, while in our approach, we never publish this information on the blockchain in the first place.

12.3.4 Self-sufficient Microgrids

Work on self-sufficient microgrids addresses the challenge of managing energy resources [93, 272, 316] through decentralized demand response algorithms [243, 326, 340] so that a microgrid's internal consumption equals its production. In contrast, we consider a local distribution grid which is part of a large hierarchical energy supply network and hence do not try to achieve a balance of zero.

12.4 System Design

In this section, we introduce a blockchain-based system design for trustless and privacy-preserving automatic netting to maximize a community's internal consumption.

Hereby, we proceed in two steps: First, we address the trust issue by proposing a blockchain-based system architecture. We add privacy protection for households through zero-knowledge off-chain computations in a second step.

12.4.1 A Blockchain-based Architecture

In the current energy market, without energy sharing in a community, consumers do not have to trust other actors. A smart meter measures the net produced or consumed energy for a household in a given time interval. The meter's balance is directly used for accounting by the supplier, which is the only trading partner. If there should ever be a dispute with the supplier, the calibrated smart meter acts as the single source of truth.

This changes when a community of households uses a netting algorithm, as described in depth in Section 12.6, to maximize their internal consumption by reducing trading volume with the supplier as much as possible. In that case, meter readings are no longer sufficient: They do not report community-internal re-allocation of energy. This information cannot be measured, because energy transfers between households are purely virtual, i.e., happen on a logical level solely for accounting purposes. The virtual transfers do not reflect physically in the local distribution grid. Due to this loss of a trusted reference point, households would not be able to dispute invalid electricity bills. Hence, the supplier can no longer be in charge of accounting. Nor can any other single party as the households mutually distrust each other.

To address this trust issue and thereby enable our netting-based approach to community-internal consumption maximization, we propose a blockchain-based system design. The core idea is to calculate the netting function in a smart contract deployed to a blockchain-network formed by households in a local distribution grid and the supplier. This design ensures decentralized, censorship-resistant, and agreed-on processing that does not require trust among the participants. Together, the calibrated electricity meters and our blockchain-based netting system re-establish a trusted reference point in a world with peer-to-peer energy sharing.

As depicted in Figure 12.3, the architecture comprises three main components: smart meters, household processing units, and a blockchain supporting smart contracts. The process of performing a trustless netting in the network is as follows:

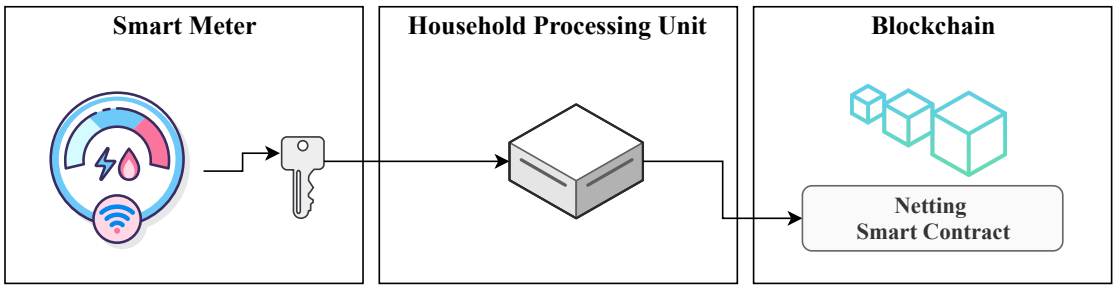


Figure 12.3: Conceptual Architecture without Privacy Protection.

The *smart meter* periodically measures the energy consumption and production within a household through a set of internal sensors. It aggregates the sensor data to a net production value for a time interval h_t, i.e., the inputs to the netting Algorithm. h_t is measured in energy units, e.g., kWh or Ws, and has positive values for producers and negative values for consumers. The smart meter signs h_t and sends it to the household processing unit.

Acting as a bridge, the *household processing unit (HPU)* receives $(h_t, sig(h_t))$ from the smart meter and registers it in the netting smart contract through a blockchain transaction. As the reported data is signed by the smart meter, the HPU cannot manipulate it without detection through the receiving smart contract. Being able to participate in the netting process provides a strong economic incentive always to report data. Withholding is possible but leads to opportunity costs.

The *blockchain* is formed by the households and the supplier in a local distribution grid. It contains a *Netting Smart Contract* that is responsible for tracking energy consumption and execution of the *netting algorithm* N, which is triggered periodically and calculates $N(\vec{h}_t)$ (see Section 12.6). After a netting has successfully been performed on the blockchain, the HPU retrieves its netting result $N(h_t)$ and the set of virtual transfers the household was involved in.

12.4.2 Adding Privacy

While successfully automating the netting process and addressing the trust challenge, the system design introduced in the previous section suffers from weak privacy guarantees. Potentially very sensible consumption data is shared between all participants in the blockchain network. All households can see each other's consumption due to the transparent processing within the Netting smart contract. Hence, in a second step, we improve on this system by strengthening households' privacy guarantees through leveraging zero-knowledge verifiable off-chain computations as described in Chapter 5. As explained in detail in Part II of this thesis, this approach allows information to be used in off-chain processing without revealing it on the blockchain. We leverage this capability in the design subsequently introduced.

To ensure privacy, households' consumption data must never be published on the blockchain. The netting algorithm, however, does require access to that data. There seems to be a fundamental conflict: The netting needs to be calculated in a trustless way — which motivated a blockchain-based design in the first place — and at the same time, the data necessary cannot be written to the blockchain.

We resolve this conflict by introducing a new blockchain-external component, the *netting entity*, which executes the netting algorithm as a zero-knowledge verifiable off-chain computation. This modification allows the blockchain to verify that the computation happened correctly without requiring access to sensitive data. Still, the households need to be convinced that the correct input data was used for the off-chain computation, and the result they learn from the netting entity is the actual output and not some arbitrary data. We address this by binding participants to input and output values on the blockchain through cryptographic commitments [51, 295].

The architecture resulting from this extension is shown in Figure 12.4. Subsequently, we describe this novel privacy-preserving trustless netting process:

As in the previous design, the *smart meter* provides signed net production values $(h_t, sig(h_t))$ to the HPU. Additionally, it now calculates $(comm(h_t), sig(comm(h_t))$, a signed commitment to these values and also sends it to the HPU.

The *HPU*, on receipt, forwards the signed production values $(h_t, sig(h_t))$ to the new netting entity. It publishes $(comm(h_t), sig(comm(h_t))$, the production value commitments, to the NettingVerification smart contract on the blockchain.

The new core component, the *netting entity*, contains a program which can be executed as a zero-knowledge verifiable off-chain computation and comprises the following three steps:

1. Calculate a netting result $N(\vec{h}_t)$ for the inputs $\vec{h}_t$ by executing a netting algorithm, but

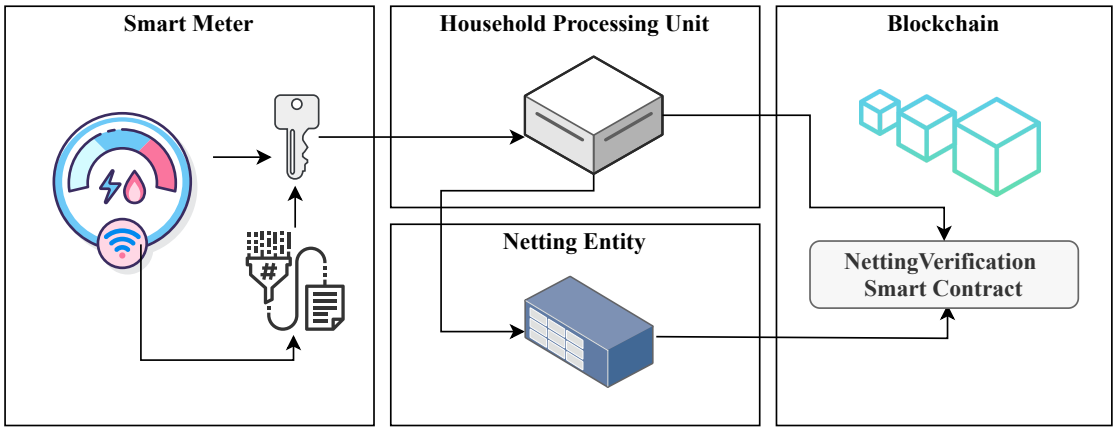


Figure 12.4: Conceptual Architecture with Privacy Protection.

keep it private through the zero-knowledge property.

2. Calculate the commitments for the inputs $comm(\vec{h}_t)$.
3. Calculate the commitments for the netting results $comm(N(\vec{h}_t))$.

Executing this program produces the output $o_t = (\pi, comm(\vec{h}_t), comm(N(\vec{h}_t)))$, where π is the cryptographic attestation of correctness. After receiving production values $\vec{h}_t$ from the HPUs, the netting entity runs the program. The resulting output o is sent to the NettingVerification smart contract.

The *blockchain* hosts a *NettingVerification smart contract*. This contract is no longer responsible for executing a netting algorithm directly. Instead, it stores commitments to inputs $comm(h_t)$ and verifies the correctness of netting computations by the netting entity. This verification comprises three steps:

1. Verify the cryptographic attestation π.
2. Verify the input commitments $comm(\vec{h}_t)$, i.e., check that the inputs to the netting algorithm used by the netting entity were the actual net production values previously committed to by the HPU.
3. Store the commitments to outputs $comm(N(\vec{h}_t))$.

To learn about the netting result, the HPU retrieves $N(h_t)$ and its virtual trades from the netting entity. It recalculates the commitment $comm(N(h_t))$ and compares it to the value stored in the NettingVerification smart contract. If equal, the HPU is convinced that the netting entity reported the netting result honestly.

This proposal enables the trustless and, at the same time, privacy-preserving netting of energy production in a local distribution grid. If the netting process fails for some reason, the system defaults to today's behavior where the supplier sells and purchases all energy. Note that while the netting entity is not trusted regarding the correctness of the netting, it is trusted with regards to

privacy: A malicious netting entity could leak consumption data. This concern can be addressed by having the meter operator, which has access to all smart meter data by design, run the netting entity.

12.5 ZoKrates-based Implementation

In this section, we describe our proof-of-concept implementation of the system proposed in Section 12.4. The overall architecture is depicted in Figure 12.5. Our ZoKrates-based prototype is open-source and available on GitHub[3].

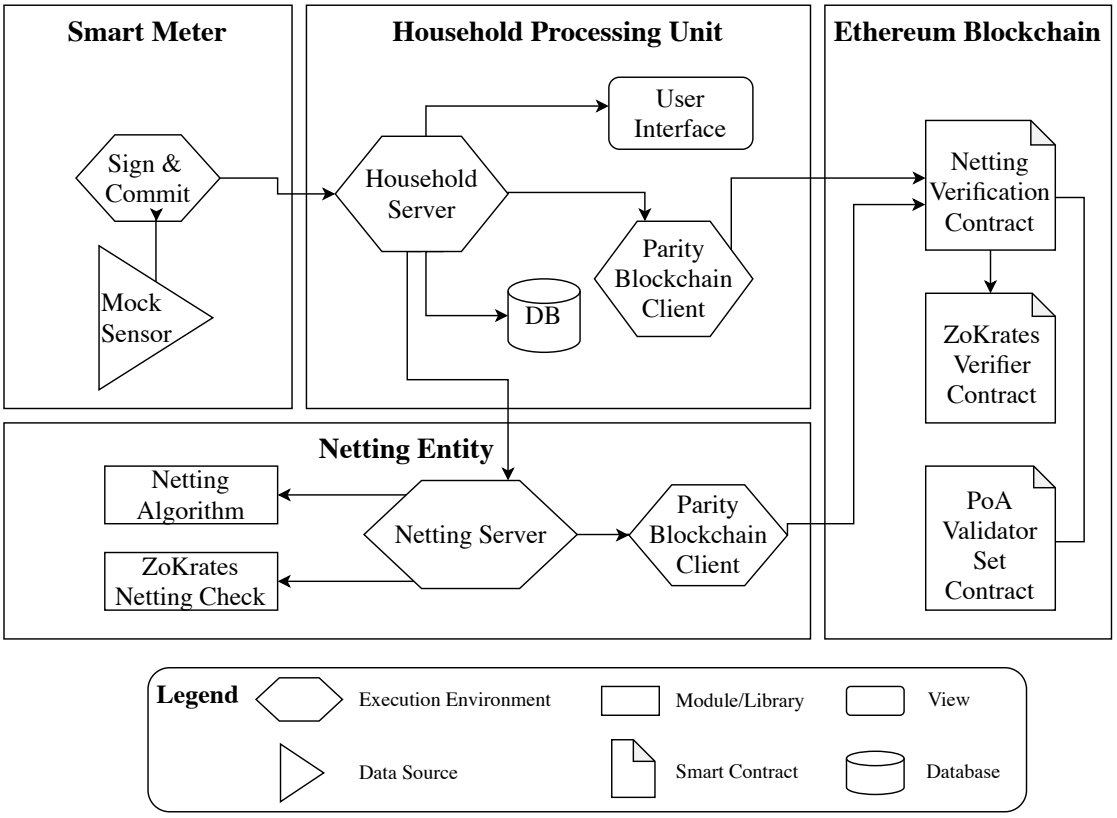


Figure 12.5: Proof-of-Concept Implementation Architecture.

12.5.1 Smart Meter

In our prototype, we simulate the *smart meter* component through a lightweight `Node.js`[4] daemon. This allows testing and benchmarking of other components without depending on physical meters. A *mock sensor* draws data from a load profile generator[5] which realistically

[3] `https://github.com/JacobEberhardt/decentralized-energy-trading`

[4] `https://nodejs.org`

[5] `https://github.com/loadprofilegenerator/automation`

simulates energy consumption for households [274]. Signatures of and commitments to the data are calculated and then forwarded to the HPU together with the raw data. As a commitment scheme $comm$, we chose $comm(v) = sha256(v||r)$, where v is the value committed to and r is a random number.

12.5.2 Household Processing Unit

Our implementation of the *household processing unit* comprises four sub-components: a household server, a database, a user interface, and a blockchain client. The implementation of the HPU is designed to run on resource-constrained devices, e.g., on Raspberry Pis that are attached to the smart meters in the *BloGPV*[6] research project's field test environment. Built with `Node.js`[4], the household server's main tasks is communication with the netting entity and the blockchain according to the description in Section 12.4. Additionally, it serves a *user interface* for end-users, which displays energy consumption over time, lists peer-to-peer transfers, and shows relevant network statistics. We provide a screenshot in Figure 12.6. The user interface is implemented with `React`[7] and the displayed data is retrieved from the household server through a REST API, where it is persisted in a MongoDB[8] instance. To connect the HPU to the blockchain, we chose a Parity Ethereum client[9].

12.5.3 Netting Entity

The *netting entity* consists of a *netting server* implemented in `Node.js`[4] that triggers the netting algorithm and invokes a ZoKrates netting check, as well as a Parity Ethereum client.

Here, the implementation is slightly different from the design introduced in Section 12.4: Instead of directly encoding the netting algorithm (see Algorithm 2) in a ZoKrates program, we run the algorithm in a separate *netting algorithm* component implemented in `JavaScript`. In a second step, a ZoKrates program called *ZoKrates netting check* checks and proves that *consistency*, *pareto efficiency* and *proportional fairness* as introduced in Section 12.6 hold for the previously calculated netting result. To account for rounding in the netting algorithm implementation, we tolerate an error ϵ for the *proportional fairness* property. We do this for efficiency, as ZoKrates programs involve overhead: Computing a netting is more complex than asserting a set of desirable properties for a given netting result. However, checking and proving these properties is sufficient, so that the netting itself can be computed in a low overhead execution environment. This approach is similar to how NP-complete problems are described: an NP-complete algorithm is hard to compute, but a given solution is easy to verify. A further advantage is that this design allows the netting algorithm to be changed without any other modifications in the system, as long as the invariants hold.

[6] `https://blogpv.net/`

[7] `https://reactjs.org/`

[8] `https://www.mongodb.com`

[9] `https://www.parity.io/ethereum/`

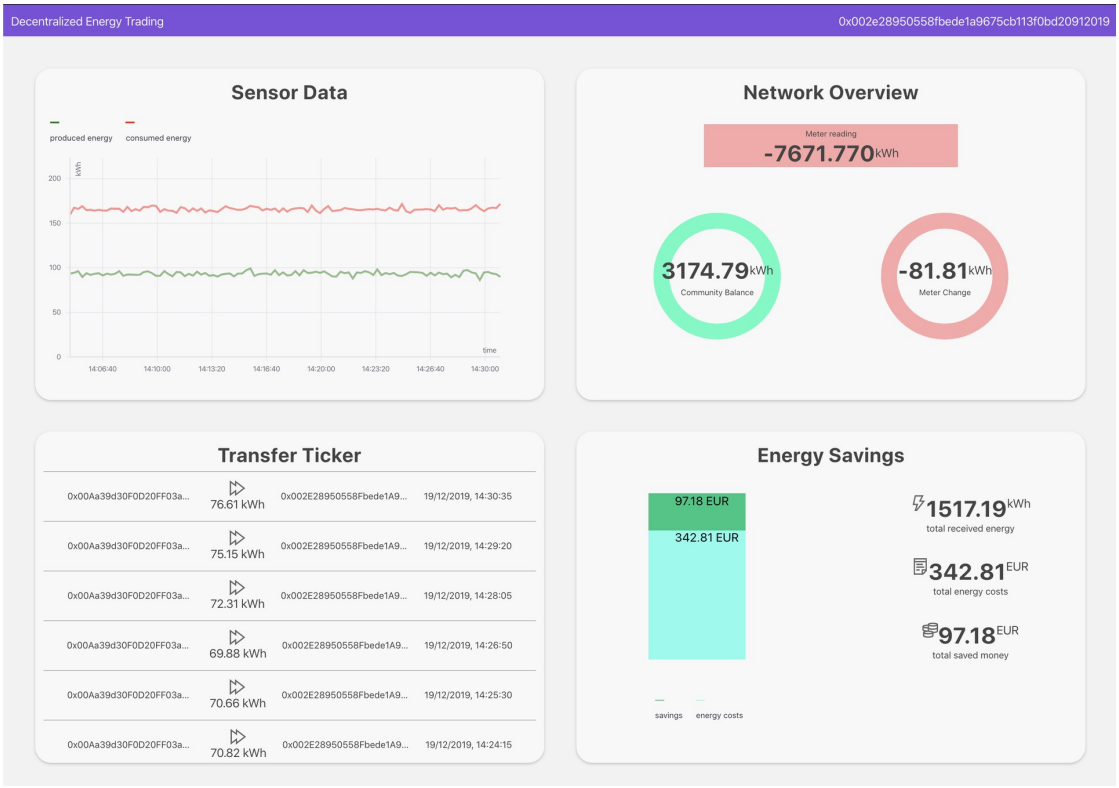


Figure 12.6: User Interface presented to a Household.

12.5.4 Blockchain

For our prototype, we chose Ethereum as an established state-of-the-art smart contract–enabled blockchain. Furthermore, ZoKrates natively supports Ethereum and generates smart contracts for proof verification. We assume the Ethereum blockchain to be deployed as a private network that connects households and the supplier in a local distribution grid. As a consensus algorithm, we chose Proof-of-Authority (PoA). In our deployment, calibrated smart meters represent authorities and sign transactions on behalf of HPUs which run blockchain clients. These authorities are registered in the *PoA Validator Set contract* that stores a list of validators allowed to sign blocks as part of the PoA Consensus. In this setup, each household independently validates and votes on all blockchain transactions, thereby establishing a trustworthy reference point. Due to the close proximity of nodes in a local distribution grid in combination with PoA, high transaction throughput can be achieved, i.e., block intervals can be short and block gas limits high. The registry of smart meters in the local distribution grid in the PoA Validator Set contract serves a second purpose: It allows the NettingVerification contract to ensure that commitments to household net production values actually come from a calibrated smart meter. The *ZoKrates verifier contract* is generated by the ZoKrates toolchain and acts as a library that supports checking an attestation of netting correctness' validity. It is called by the *NettingVerification contract* as introduced in Section 12.4.

As a summary, we provide an overview of the netting process in Figure 12.7.

12.6 Netting

To incentivize the production of renewable energies, we propose a community-oriented trading model where households maximize the use of energy within their community. This re-allocation among households minimizes the amount of energy traded with utilities. A supplier only sells or buys the residual load, i.e., the net amount of energy prosumed in a local distribution grid. Hence, we refer to this approach as *netting*.

Unlike related work (see Section 12.3.1), which commonly seeks to bring together future demand and supply, we take an ex-post perspective: For a given time interval, energy is produced and consumed. After the interval passed, we observe the amount of energy that was produced and consumed within the local distribution grid. Based on that data, we calculate a netting, which introduces transfers that happen within a community and thereby minimizes the energy traded with the supplier. Essentially, this approach only affects accounting. The physical energy flow is not affected.

Compensation for the transfers within a community can happen but does not have to. Households are never worse off after a netting, so they may be satisfied by occasionally receiving and giving away energy for free.

12.6.1 Characterizing Desirable Netting Results

As described before, the netting process comprises the creation of virtual transfers between households so that only residual load is traded with the supplier. However, there are many possible

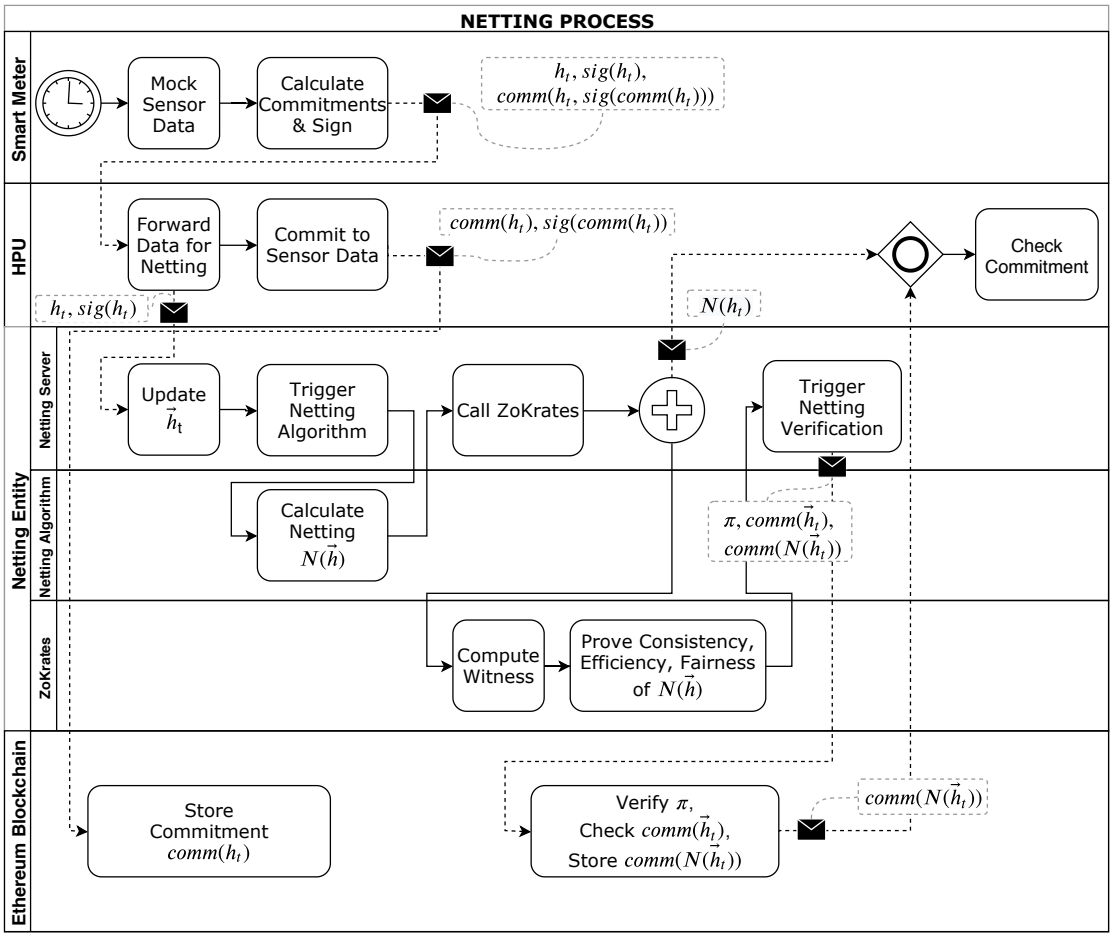


Figure 12.7: Overview of the Netting Process.

ways to allocate these energy flows to households in the network. In this section, we define desirable properties of netting results and formulate characterizing invariants. These properties should be considered during the design of netting algorithms.

Let $\vec{h} = \Big(h_1, h_2, \ldots, h_n\Big)^T \in \mathbb{R}^n$ be the grid energy trading balance as a vector of net energy sales of $n \in \mathbb{N}$ households to the supplier for a given time interval. These values are denoted in energy units, e.g., kWh or Ws, and are positive for sellers and negative for buyers. The total amount of energy bought from the supplier $e_b \colon \mathbb{R}^n \to \mathbb{R}$ and the total amount of produced energy sold to the supplier $e_s \colon \mathbb{R}^n \to \mathbb{R}$ are defined as:

$$e_b(\vec{h}) = \sum_i^n min(h_i, 0), \quad e_s(\vec{h}) = \sum_i^n max(0, h_i)$$

For convenience, instead of writing $e_b(\vec{h})$ and $e_s(\vec{h})$, we write e_b and e_s. We define a netting algorithm as a function $N \colon \mathbb{R}^n \to \mathbb{R}^n$, which calculates the new and reduced net trading volume

with the supplier $\vec{h}_{net}$.

Let N be an arbitrary netting algorithm. We now define the invariants as follows:

Consistency

After a successful netting, the total energy balance within the grid remains unchanged, i.e., energy is never artificially lost or introduced into the system.

$$e_b(\vec{h}) + e_s(\vec{h}) = e_b(N(\vec{h})) + e_s(N(\vec{h}))$$

While this invariant is the weakest of all, it is also the most fundamental which characterizes all valid netting algorithms.

Pareto Efficiency

No household should ever be worse off after participating in a netting. This property is captured through the following invariant:

$$|N(\vec{h})_i| \leq |\vec{h}_i| \quad \forall i \in \{1, \ldots, n\}$$

Pareto efficiency indicates that after a successful netting, no producing household sells more energy to the supplier, and no consuming household buys more energy from the supplier than before.

Fairness

Fairness characterizes how the benefits that can result from netting should be allocated to participating households. There are many different ways to define fairness, e.g., lower participation threshold or equal distribution. We introduce the notion of *proportional fairness* as a specific type of fairness that suits our use case.

In a proportionally fair netting, the benefits gained from minimizing trading volume with the supplier are allocated to households in proportion to their prosumption. We distinguish three cases depending on the relation between e_b and e_s:

1. $e_b > e_s$: In this case, households consumed more energy than they produced. All energy produced by households is allocated to consuming households in proportion to their consumption:

$$N(\vec{h})_i = \begin{cases} h_i + e_s \cdot \frac{h_i}{e_b}, & h_i < 0 \text{ (consumer)} \\ 0, & \text{otherwise} \end{cases}$$

2. $e_b < e_s$: In this case, households produced more energy than they consumed. All consumed energy can be supplied from within the community directly, and the residual production

is sold to the supplier. Here, the share a producer transfers to consuming households is proportional to its share of the overall production:

$$N(\vec{h})_i = \begin{cases} h_i + e_b \cdot \frac{h_i}{e_s}, & h_i > 0 \text{ (producer)} \\ 0, & \text{otherwise} \end{cases}$$

3. $e_b = e_s$: In this unlikely case where consumed and produced energy are equal, the condition collapses to the trivial case, where no energy is traded with the supplier at all after netting:

$$N(\vec{h})_i = 0 \quad \forall i \in \{1, \ldots, n\}$$

The above conditions directly imply the following invariants:

$$e_s(\vec{h}) > |e_b(\vec{h})| \Rightarrow e_b(N(\vec{h})) = 0$$
$$e_s(\vec{h}) \leq |e_b(\vec{h})| \Rightarrow e_s(N(\vec{h})) = 0$$

For a netting result which fulfills *proportional fairness*, either energy bought by consuming households or energy sold by producing households will be zero.

12.6.2 A Fair and Constructive Netting Algorithm

After characterizing desirable netting results, we now introduce an algorithm to compute a netting that fulfills these properties. This algorithm reduces the amount of energy traded with the supplier and instead maximizes trading volume between households. It is *constructive*, i.e., it computes and records virtual trades between households in the process.

Algorithm 2 computes a netting result which fulfills *consistency*, *efficiency*, and *proportional fairness* up to a rounding error ϵ. Let $\vec{h} = (\vec{h_s}, \vec{h_b})$ be the grid energy trading balance ordered by selling and purchasing households, and e_s, e_b the total amount of energy sold to and bought from the supplier, as defined in Section 12.6.1.

The computed netting function $N \colon \mathbb{R}^n \to \mathbb{R}^n$ reassigns the new energy balances of each household as a state transition:

$$N(\vec{h}) = \begin{cases} min_i(h_i + \left\lfloor e_s \cdot \frac{h_i}{e_b} \right\rfloor, 0 + \frac{\epsilon}{n}), & \text{if } |e_b| \geq e_s \\ \\ max_i(0 + \frac{\epsilon}{n}, h_i + \left\lfloor e_b \cdot \frac{h_i}{e_s}) \right\rfloor, & \text{otherwise} \end{cases}$$

where min_i and max_i are the element-wise min and max functions.

Intuitively, the algorithm transfers energy that has been produced in a community to consuming households in proportion to their consumption.[10] For that, it iterates through the set of producing

[10]And vice-versa in the case where the community produces more energy than it consumes.

Algorithm 2 Proportional Netting

procedure NET(e_s, e_b, $\vec{h}_s$, $\vec{h}_b$)
  if $|e_s| < |e_b|$ **then**
    $\vec{h}_{from}, e_{from}, \vec{h}_{to}, e_{to} \leftarrow \vec{h}_s, e_s, \vec{h}_b, e_b$
  else
    $\vec{h}_{from}, e_{from}, \vec{h}_{to}, e_{to} \leftarrow \vec{h}_b, e_b, \vec{h}_s, e_s$

  for all $h_t \in \vec{h}_{to}$ **do**
    $e_{allocate} \leftarrow \left\lfloor e_{from} * \frac{h_t}{e_{to}} \right\rfloor$
    for all $h_f \in \vec{h}_{from}$ **do**
      if $e_{allocate} \neq 0$ **then**
        if $|e_{allocate}| \leq |h_f|$ **then**
          $h'_f, h'_t \leftarrow transfer(h_f, h_t, e_{allocate})$
          $e_{allocate} \leftarrow 0$
        else
          $h'_f, h'_t \leftarrow transfer(h_f, h_t, h_f)$
          $e_{allocate} \leftarrow e_{allocate} - h_f$
        $\vec{h}_{from}[index(h_f)] \leftarrow h'_f$
        $\vec{h}_{to}[index(h_t)] \leftarrow h'_t$
  return $\vec{h}_{from}, \vec{h}_{to}$
procedure TRANSFER(h_{from}, h_{to}, e)
  record $(index(h_{from}), index(h_{to}), e)$
  $h_{from} \leftarrow h_{from} - e$
  $h_{to} \leftarrow h_{to} + e$
  return h_{from}, h_{to}

households. Each household transfers its energy to entitled consumers. As an example, Figure 12.8 shows a step-wise computation of a constructive and fair netting using the proposed algorithm. The energy that has been produced in the community and would have originally been sold to the supplier is allocated to consumers and thus reduces the amount of energy they have to purchase.

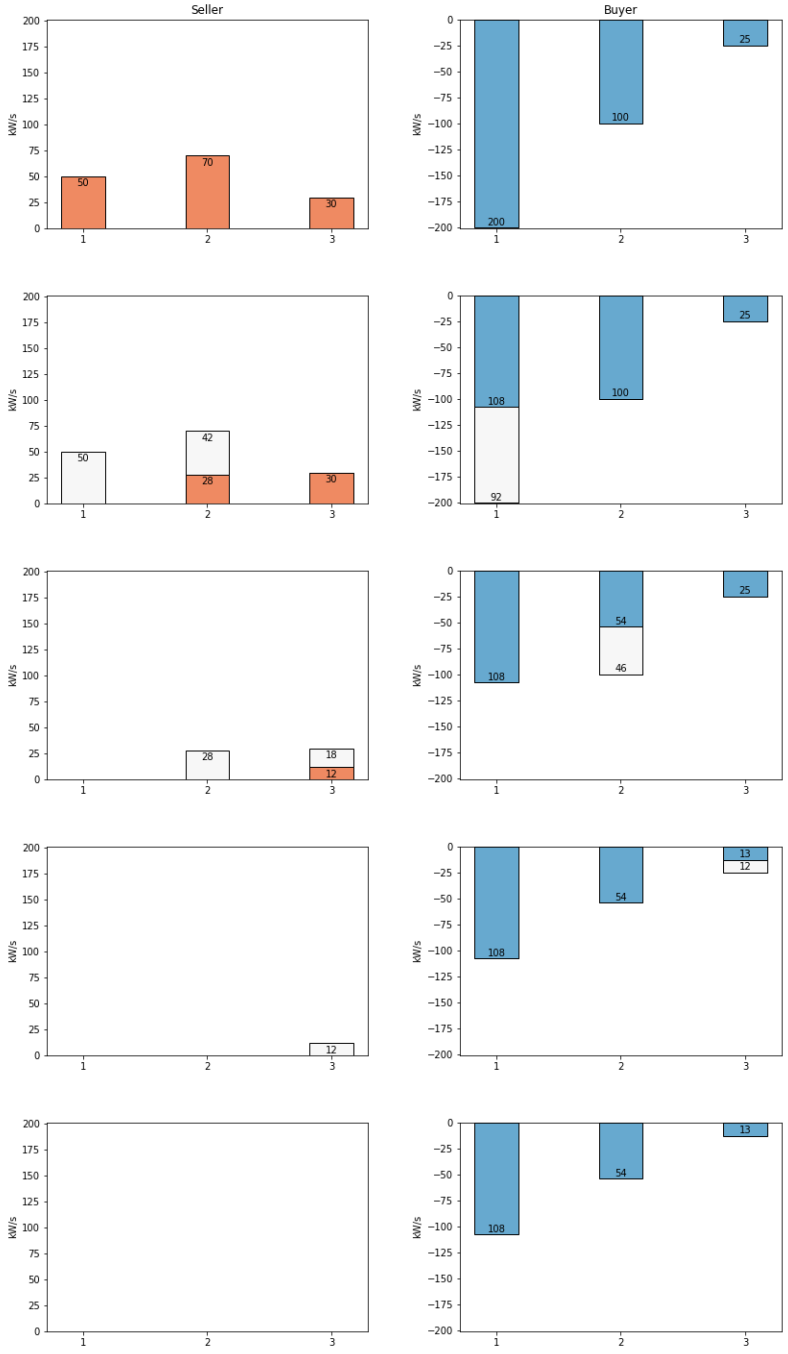


Figure 12.8: Example Sequence of the Netting Algorithm

12.7 Performance Evaluation

In this section, we present a performance evaluation of our ZoKrates-based implementation of the previously introduced system. Our evaluation was conducted in the context of a field test within the *BloGPV* research project.

The field test targeted a netting interval of 15 minutes and a local distribution grid comprising 100 households. This netting interval defines the upper bound for the time it can take to calculate the netting and then prove its *correctness*, *pareto efficiency*, and *fairness* in a ZoKrates program within that netting interval. The other components implemented as part of our prototype are very lightweight and fulfill their tasks for a netting interval within microseconds, even in resource-constrained environments. Therefore, we limit our performance evaluation to the netting entity. Here, we focus on ZoKrates-related steps, since the execution of the netting algorithm completes in microseconds and is hence negligible.

In Figure 12.9, we show benchmarks for program execution and proof generation time. The benchmarks were performed on a consumer laptop with an Intel Core i7-6920HQ @2.9 GHz processor, 16 GB RAM, and a 1 TB SSD. We followed the experiment plan described in our performance evaluation in Section 11.1. Together, program execution and proof generation took ~14 min on average without optimizations. The standard deviation between experiment runs was negligible, i.e., $\sigma < 1\%$, and thus not reported explicitly. Hence, the overall netting time stays below the set 15-minute goal. Thus, we leave optimizations for future work. We expect significant speedups from optimizations in the ZoKrates code, e.g., the use of Pedersen commitments [270] instead of our SHA-256-based commitment scheme. Furthermore, the process could be accelerated through more powerful hardware.

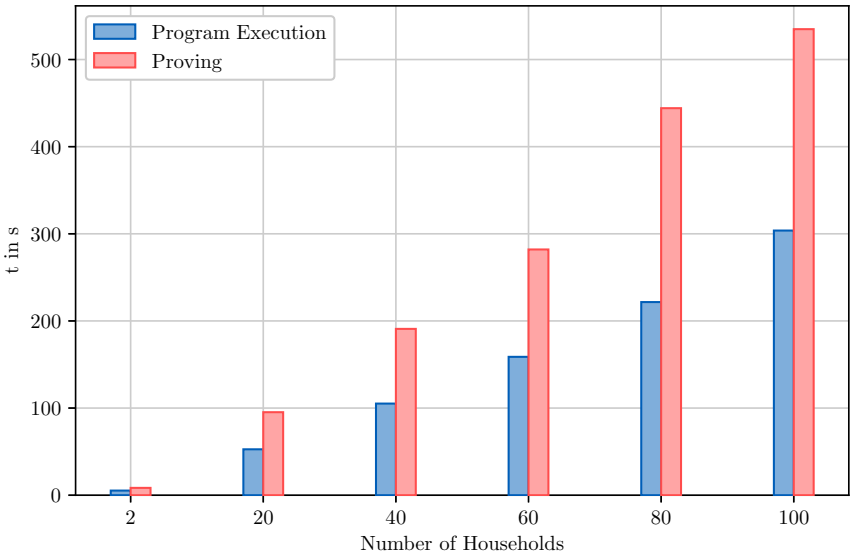


Figure 12.9: Program Execution and Proving Time for the ZoKrates Program.

We provide an overview of the gas cost required for the verification of a netting result in the NettingVerification smart contract in Figure 12.10. As the private blockchain employed in the field test is formed by participants of the local distribution grid with low network latencies, the required gas-throughput is easily achieved.

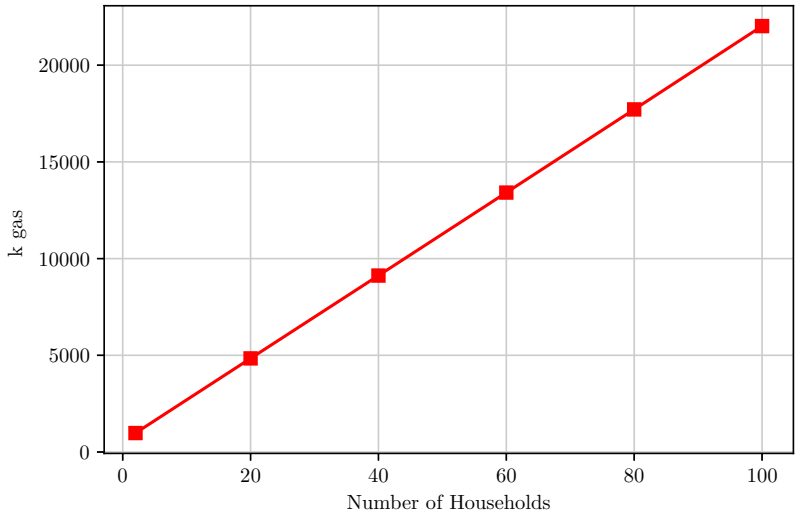


Figure 12.10: On-chain Verification Cost of a Netting Result.

12.8 Conclusion

In this chapter, we demonstrated the viability of ZoKrates as a privacy-engineering tool in the context of decentralized applications. We introduced a blockchain-based and privacy-preserving system design for energy sharing within a community through ZoKrates-based verifiable off-chain computations. Then, we described our open-source proof-of-concept implementation and its extensive evaluation within the *BloGPV* research project. More generally, we showed how ZoKrates-based off-chain computations can be combined with on-chain commitments to execute algorithms in a group of distrusting members with blockchain properties while preserving privacy.

CHAPTER 13

Scalable Blockchain Relay

In this chapter, we demonstrate how ZoKrates-based off-chain computations can be used to allow one blockchain to validate data and events of another blockchain in an efficient and scalable way. We introduce a relay design that moves header validation off-chain through verifiable off-chain computations and thereby reduces the cost of validating block headers of a Proof-of-Work (PoW) source blockchain on a target blockchain. As a proof-of-concept, we provide a ZoKrates-based implementation for a Bitcoin relay on the Ethereum blockchain that allows validating 504 Bitcoin headers in a single Ethereum transaction. Compared to BTC Relay, the verification cost is reduced by a factor of 187. In the context of this thesis, this application demonstrates that ZoKrates-based off-chain computations represent a powerful abstraction to address scalability concerns.

The material presented in this chapter is based on and contains material from our research paper co-authored with Martin Westerkamp that was published at the IEEE Security & Privacy on the Blockchain (IEEE S&B) 2020 workshop [343]. In this chapter, we provide an overview of our design and the ZoKrates-based realization. For further details and an extended discussion, we refer the reader to our paper. The zkRelay implementation is open source and available on GitHub[1].

13.1 Motivation and Overview

Different trust assumptions, intended use cases, and community goals lead to the development and deployment of multiple blockchain networks in recent years. Yet these blockchains — including their tokens and applications built on top — exist in isolation. Connecting them could unlock network effects, reduce lock-in, and speed up innovation. For example, assets would no longer be tied to one blockchain, and cross-chain smart contracts could react to events on several chains or be migrated to the most suitable network [342].

[1] `https://github.com/informartin/zkRelay`

Links between blockchains can be unidirectional or bidirectional. A blockchain that can read a second blockchain is referred to as a *sidechain* of that chain [22]. The chain that is read from is commonly called *parent chain* [359]. Sidechains can access and react to events, e.g., transactions or state changes, on linked parent chains. *Pegged sidechains* are bidirectionally linked blockchains that support asset transfers [73].

From a technical perspective, blockchain interoperability represents a significant challenge: The connection between blockchains must be designed in a way that introduces as little trust as possible to not compromise their key properties. Two generic approaches to establish a trustworthy link between blockchains have been proposed: trusted intermediaries and relays [73].[2]

Trusted intermediaries, in the form of a single trusted entity or a trustworthy group, can act as notaries that mediate between blockchains. They facilitate blockchain interoperability by publishing signed source chain information to a target blockchain, which in turn reacts to it after validating corresponding signatures. Oracle protocols try to reduce the trust required in notarizing intermediaries, for example through economic incentives and voting [173].

Blockchain relays, in contrast, establish a unidirectional link without needing to trust intermediaries: A relay is a component on a target blockchain that directly validates the consensus rules of a source blockchain. Thus, arbitrary untrusted parties can publish block headers to chain relays. The on-chain relay acts as a light client that checks the correctness of its source ledger's header chain. Using headers from that chain, external users can then create simplified payment verification (SPV) proofs [258] to demonstrate that an event occurred on the source ledger or that some data is part of its state. SPV proofs are usually Merkle proofs [241] that prove inclusion of data in a Merkle tree that is anchored in a block header via its root hash.

Current relay implementations, e.g., BTC Relay, a blockchain relay from Bitcoin to Ethereum [67], sequentially receive and validate blocks from a PoW source blockchain based on its consensus rules. By doing so, they re-create the source blockchain's header-chain on the target blockchain. Due to this design, the cost of submitting an up-to-date block header grows with the distance to the last block that has been submitted to the relay. Before SPV proofs can be created for an up-to-date block, all intermediate block headers must be published and validated. This overhead leads to significant problems in practice, despite fee models that have been proposed to incentivize external relayers to publish block headers to the on-chain relay to ensure liveness. The BTC Relay contract, for example, has not validated a block for more than two years, which corresponds to over one million Bitcoin blocks at the time of writing. Consequently, over one million transactions of ~194,000 gas each would be required to catch up with the Bitcoin network.[3] It is unlikely that any network participant is willing to pay the associated accumulated transaction fee of ~288,000 €.[4]

To address this issue, we propose a relay system that leverages ZoKrates-based verifiable off-

[2] In [73], Buterin mentions hash-locking as a third, more specialized interoperability approach for atomic cross-chain exchange. We do not consider this approach due to its limited scope.

[3] An example of a Bitcoin header submission to the BTC Relay contract can be found in the Ethereum transaction with the following ID: 0xe21099d8fd1252281389fc888f23f98e60db22ecb5c149ad6fda6dccdf110b50

[4] Estimations based on network statistics observed on the Ethereum mainnet on 01.05.2020.

chain computations to prove the validity of a batch of block headers for a PoW source blockchain. A relay smart contract on the target blockchain can then verify the resulting short and constant-size proof in a single on-chain transaction to confirm the validity of the whole batch. For each batch, only a single block is stored on the target ledger. Importantly, our design does not compromise the ability to perform SPV for batched blocks.

In the next section, we introduce our conceptual relay design. Afterward, we describe our ZoKrates-based implementation that realizes a bridge between Bitcoin and Ethereum. We demonstrate a cost reduction for Bitcoin block verification by a factor of 187 compared to BTC Relay by batching 504 bitcoin headers in a single zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) proof. Although our implementation is specific to Bitcoin as a source and Ethereum as a target, it can be easily adapted to support arbitrary PoW source blockchains and any target blockchain capable of zk-SNARK verification.

13.2 Conceptual Design

The goal of a relay system is to allow a target ledger L_T to read and react to events on a source ledger L_S.[5] In this section, we describe the design of zkRelay, a relay system that supports batched validation of header chains by leveraging verifiable off-chain computations.

We proceed in four steps, where each step solves a specific design problem and builds upon the previous ones: First, we describe how a single block header can be validated in an off-chain computation and discuss PoW-related peculiarities that the relay smart contract on the target blockchain needs to account for besides proof verification. Second, we extend the off-chain validation approach to epoch-sized header batches. Third, we describe how SPV can be enabled even for intermediary blocks that are not published on-chain before introducing support for flexible batch sizes in a last step.

Our relay system consists of a zkRelay client and a zkRelay smart contract. The zkRelay client mediates between source and target ledgers by running off-chain header chain validation programs and generating correctness proofs. The zkRelay smart contract validates these proofs and accepts associated block headers. A high-level overview of this process and involved components is depicted in Figure 13.1.

13.2.1 Off-chain Validation of a Single Block Header

In this section, we describe a basic version of zkRelay that validates single headers in off-chain computations without batching. This allows us to discuss the handling of intricacies of PoW blockchains, i.e., difficulty adjustments, the lack of finality, and the occurrence of forks, independently from off-chain batch validation.

The off-chain validation logic is specified in a ZoKrates program. This program receives a block header as private input and its claimed header hash as public input and returns a boolean value

[5] To avoid confusion between batches (B) and blockchains, we use the term ledger (L) to refer to the latter for notational convenience.

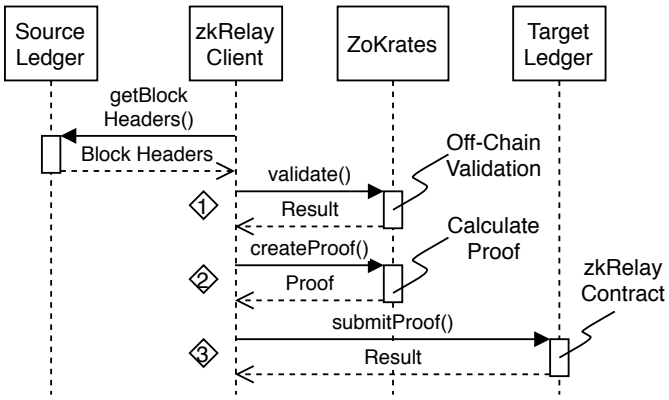


Figure 13.1: Overview of the zkRelay System

that indicates the validity of the header's PoW condition.

On deployment, the zkRelay contract is initialized with the genesis block $\mathcal{G}$, i.e., $X = 0$ (or a more recent agreed-on block) of the source blockchain L_S. Then, to synchronize the relay contract with the source chain up to a target block, zkRelay queries L_S to retrieve the chain of related block-headers. We denote a header as H_S^X, where X refers to the height of the corresponding block on L_S. This corresponds to step 1 in Figure 13.1. Afterward, steps 2 and 3 are performed sequentially for each of these headers: A proof attesting to the correctness of the PoW condition is generated and submitted to the zkRelay contract. This contract validates the proof, checks whether the predecessor H_S^{X-1} referenced by the submitted header H_S^X is part of its local header chain and updates this chain in case of success.

Difficulty adjustments: PoW blockchains periodically adjust mining difficulty to achieve relatively constant block intervals even in case of changes of the network's overall hash power. Difficulty adjustments reflect in the PoW condition that has to be satisfied by all valid blocks: Their header hashes have to be smaller than a difficulty-dependent target value. The target difficulty is recalculated for every epoch E which consists of L blocks.

Since the difficulty target for a block is encoded in its block header, it can be considered in the off-chain PoW condition check. The validity of the encoded difficulty target itself, however, cannot be checked in our off-chain program. Instead, the zkRelay contract checks the encoded difficulty target as part of the on-chain validation step 3: For H_S^X with $X \bmod L = 0$, i.e., headers of the first block of an epoch, the target has to be calculated from the timestamps of the first and the last blog of the preceding epoch. For all other headers, the encoded target has to equal that of its predecessor.

Fork handling: Due to the probabilistic nature of PoW consensus, a valid header received by the zkRelay contract is not guaranteed to be and remain part of L_S; it could also be part of a fork. Thus, the zkRelay contract tracks parallel forks and selects the fork with the most accumulated PoW as the main header chain.

Finality: PoW blockchains offer a probabilistic finality guarantee. The probability that a block is invalidated by a blockchain reorganization becomes increasingly unlikely with the number of succeeding blocks. This property is inherited by the zkRelay system and has to be accounted for by external users that rely on the zkRelay header chain for SPV. To prevent the construction of SPV proofs on forks, we recommend that external smart contracts that reference the zkRelay contract introduce a security parameter n which denotes the minimum number of validated successors a given header must have to be considered final. Due to the inherent tradeoff between latency and security, suitable values for n depend on the needs of consuming applications.

13.2.2 Off-chain Validation of Epoch Batches

In this section, we extend our relay design to support the off-chain validation of batches of headers of an epoch. Not publishing intermediary headers significantly reduces the on-chain verification and storage cost. Using this approach, the cost of synchronizing stalled relays can be reduced to a fraction.

A batch B_N is defined as an ordered list of N block headers $H_S^{X..(X+N-1)}$. The goal of off-chain batch validation is to check the validity of a header chain in a single off-chain computation and to only publish the last header of that chain to the relay smart contract. We call the block headers that are not published on-chain intermediary headers.

To support batch validation, our ZoKrates off-chain validation program has to be extended. For a given input B_N, it has to check the validity of the header chain, the PoW condition for each header, and the correctness of the used difficulty targets.

For each epoch E, a difficulty target is computed from the difference in timestamps Δt between the first and last blocks of the previous epoch, the prior difficulty target, the epoch length L, and the target block interval θ:

$$target_E = \frac{\Delta t}{\theta \times L} \times target_{E-1}$$

Unlike in the case of the off-chain validation of single block headers, difficulty accounting has to be performed in the off-chain validation program since the required information is hidden in intermediary headers and not available to the zkRelay contract.

We now present a solution for off-chain batch validation where the batch size is equal to the epoch length ($N = L$) and the last block of a batch coincides with the first block of an epoch. We generalize this result to arbitrary sizes of N in Section 13.2.4.

A zkRelay epoch batch comprises L headers where the last header corresponds to the first block of an epoch E and the other $L - 1$ headers are from the previous epoch $E - 1$. Since the last block of a zkRelay batch has to be stored in the zkRelay contract, it is provided to the ZoKrates off-chain validation program as public input while all other headers of a zkRelay epoch batch are provided as private inputs. The ZoKrates off-chain validation program checks the validity of this header chain as well as the validity of the PoW conditions for each contained header. As discussed before, we additionally have to validate the recalculation of the difficulty target for the new epoch that contains the last block of a zkRelay epoch batch in our ZoKrates program.

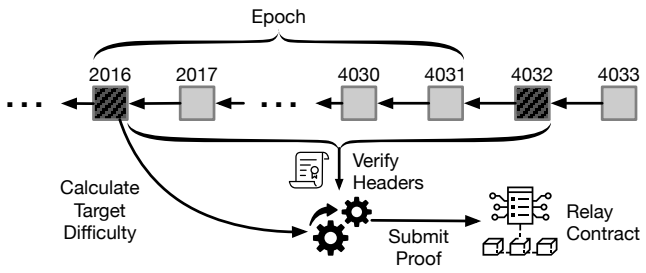


Figure 13.2: zkRelay Epoch Batch Validation ($N = L = 2016$)

While both target difficulty values ($target_E$, $target_{E-1}$) and the timestamp of the last epoch block are encoded in our inputs, the timestamp of the first epoch block required to compute Δt and evaluate the difficulty condition is missing. Thus, we additionally provide the first header of the previous epoch $E - 1$ as input and validate its PoW condition and inclusion in the header chain. We make this input public so that the zkRelay contract can check consistency with the header persisted on-chain.

An overview of this process and the relation between epochs and zkRelay epoch batches is depicted in Figure 13.2. The dark boxes represent headers that are public inputs and stored in the zkRelay contracts. The light gray boxes are intermediary headers that are not published to the blockchain.

13.2.3 Simplified Payment Verification for Intermediary Headers

Previously, we showed how epoch-sized batches of headers can be validated in a single off-chain computation. This aggregation significantly reduces the storage and processing cost for target ledger L_T. However, the ability to create SPV proofs for intermediary blocks was lost in the process, since their block headers are no longer published to the zkRelay contract. In this section, we introduce a mechanism that re-enables SPV for intermediary headers.

The fundamental idea is to construct a Merkle tree that has the header hashes of the headers in a batch as its leaves in the off-chain validation program. Besides the validation result, this program returns the root hash of that Merkle tree. This root hash is then published to the zkRelay contract and persisted for each batch. External users can then publish intermediary block headers together with Merkle inclusion proofs. The zkRelay validates these proofs against the stored root hash and adds the intermediary headers to the corresponding batch. These intermediary headers can then used for regular SPV proofs.

Since Merkle proofs can become rather large for long batches, external users can use an additional dedicated ZoKrates program to validate a Merkle proof off-chain and only publish a proof attesting to its validity together with the block header to the zkRelay contract. We discuss an instantiation of this idea in the context of our implementation in Section 13.3.

13.2.4 Flexible Batch Sizes

Previously, we introduced a zkRelay design that allows the off-chain validation of epoch-sized header batches and demonstrated how SPV can be supported even for intermediary headers.

However, two challenges remain to be solved to ensure the practicality of zkRelay:

1. Timely SPV proofs for intermediary block headers are not supported. An epoch batch has to be validated on L_t before intermediary headers can be published and used for SPV. In Bitcoin, for example, an epoch consists of $L = 2016$ blocks and has an expected duration of 14 days. For zkRelay users, this implies delays of up to two weeks.
2. Off-chain computations become increasingly expensive with larger batch sizes. To allow running zkRelay clients on consumer hardware, support for smaller batch sizes is desirable. We evaluate and discuss performance characteristics and hardware requirements of our zkRelay implementation in Section 13.3.2.

To address these challenges, we introduce a design for flexible batch sizes. This design is fully compatible and can be combined with the Merkle tree–based SPV proofs for intermediary headers.

We provide an overview of our design for flexible batch sizes, which we subsequently explain in more detail, in Figure 13.3. Again, the dark boxes represent headers available to headers that are passed as public inputs to off-chain programs and are available to a zkRelay contract on-chain. The light gray boxes represent intermediary headers that are used during off-chain validation but are not published to L_T in the validation process.

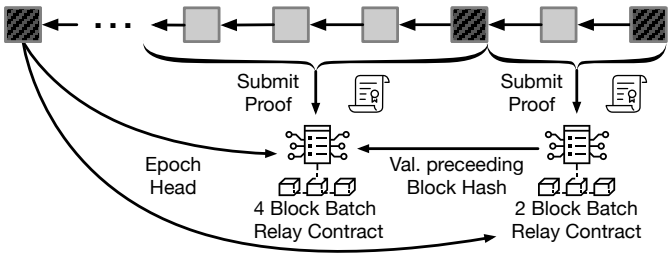


Figure 13.3: zkRelay Verification of Batches with Flexible Size

Difficulty Adjustments: With epoch-sized batches, we were able to guarantee that a new epoch is entered with every batch and that the corresponding update of the difficulty target can be verified in the off-chain computation. This guarantee no longer holds for flexible batch sizes. An epoch bound may or may not be crossed in a batch. We solve this problem in three steps:

1. First, we always provide the first block header of the epoch that contains the first header of a given batch as input to the off-chain validation program. More precisely, we provide the header $H_S^{X-(X \bmod N)}$ as public input. This ensures that all information required to validate the calculation of a subsequent epoch's difficulty target is available.

2. Then, inside the off-chain program, we check the difficulty target calculation under the assumption that the last header in the batch is that of the first block of a new epoch. We add the result of that check to the program's outputs as a boolean flag.

3. For a submitted batch, the zkRelay contract checks whether an epoch boundary was crossed. If an epoch boundary was crossed, the contract asserts that the last block in the submitted batch was the first block of a new epoch.[6] Then, it checks the correctness of the off-chain validity target calculation based on the corresponding boolean flag. The submitted batch is accepted and added to L_T if all checks pass. All batches that cross an epoch boundary by more than a single block are automatically rejected.

Accidentally surpassing epoch boundaries can be prevented by setting up zkRelay contracts in a way that batches of size N can only be created on top of headers H^X so that $X \bmod L + N \leq L$. Note that this restriction is purely intended to prevent external users from accidentally submitting invalid batches; it is not required to guarantee security.

zkRelay Contracts: A zkRelay contract stores and keeps track of the header chain of a source blockchain L_S. This header chain is extended by header batches that have been validated in an off-chain computation. The zkRelay contract confirms the correctness of the off-chain validation through verification of the respective zk-SNARK proof. There are two options for how to setup zkRelay to support flexible batch sizes up to epoch length L:

1. **One contract, any batch size.** The first option is to equip a single zkRelay contract with verification methods for all desired batch sizes up to epoch length L. This approach allows the verification of combinations of arbitrarily-sized batches within an epoch. However, the supported batch sizes have to be known and encoded into the zkRelay contract at deployment.

2. **Multiple contracts, fixed batch sizes.** The second option is to deploy multiple zkRelay contracts where each supports only a fixed batch size. To guarantee that off-chain validation remains possible when crossing epoch boundaries, this batch size has to be a fraction of the epoch length L (see previous paragraph). Support for flexible batch sizes is achieved by decomposing a given batch of arbitrary size into a set of suitably-sized sub-batches so that these sub-batches are supported by the available zkRelay contracts. Using these contracts, the sub-batches are then verified sequentially. For example, a batch of size six can be decomposed into sub-batches of size four and two. To support sequential validation, zkRelay contracts hold references to other zkRelay contracts that validate multiples of their own batch size.[7] A zkRelay contract can base its batches on headers that have been validated and stored by these referenced contracts in addition to its own headers. The advantage of this design option is that new zkRelay contracts for fixed batch sizes can be flexibly added after the initial deployment. From a practical perspective, splitting a single

[6] These two checks require the block height X, which can be provided as an output of the off-chain program.

[7] Restricting the referenced contracts to multiples of the given batch size guarantees that epoch boundaries are crossed in an orderly manner, i.e., a batch that crosses an epoch boundary contains only the header of the first block of the new epoch.

large zkRelay contract into several smaller ones can be useful to circumvent contract size limits that may apply on L_T.

The more suitable variant should be selected depending on the given use case at hand.

13.3 ZoKrates-based Implementation

In this section, we describe and evaluate our zkRelay implementation that establishes a one-way link from Bitcoin to Ethereum. We focus on ZoKrates-related aspects in the context of this thesis and refer the reader to our paper [343] for details on other components. The zkRelay source code is available on GitHub[8].

This implementation of zkRelay comprises three main components.

- **ZoKrates Off-chain Programs.** In the context of zkRelay, ZoKrates programs define the batch validation logic as well as the creation of inclusion proofs for intermediary headers as described in the previous section. They are implemented fully in the ZoKrates language.
- **zkRelay Smart Contracts.** A zkRelay contract is an Ethereum smart contract written in Solidity that validates and stores headers of a source chain L_S as previously described. For the verification of off-chain programs, it references and accesses verification smart contracts generated by the ZoKrates framework.
- **zkRelay Client.** The zkRelay client provides an entry point for users of the zkRelay system and is written in Python. It retrieves Bitcoin block headers, coordinates the execution of and proof generation for ZoKrates programs, and interacts with zkRelay contracts on Ethereum.

13.3.1 ZoKrates Off-chain Programs

Our zkRelay implementation uses ZoKrates programs for off-chain batch validation and inclusion proofs for intermediary headers. The zkRelay client supports the generation of both types of ZoKrates programs for arbitrary batch sizes.

Batch Validation: The Bitcoin protocol, i.e., the consensus rules of L_T, determines the off-chain batch validation logic. The implementation of a Bitcoin batch verifier in ZoKrates closely follows the conceptual description in the previous section. As a first step in the off-chain validation process, the zkRelay client preprocesses the previously retrieved raw Bitcoin headers and converts them into a set of inputs compatible with the ZoKrates data types. Concretely, a Bitcoin header has a size of 80 bytes [49] and is thus split into five field elements where each field element encodes 16 bytes. Inside the off-chain validation program, the SHA-256 hash function is applied twice to each header. Based on the resulting header hashes, the validity of the header chain contained in a batch is checked. Additionally, the PoW condition is validated for each header based on the encoded difficulty target. As described in Section 13.2.4, an additional validity check of the difficulty re-calculation for the last header is included.

[8] Source code available at `https://github.com/informartin/zkRelay`.

Inclusion Proofs for Intermediary Headers: In Section 13.2.3, we proposed to construct a Merkle tree over the intermediary headers of a batch to enable Merkle inclusion proofs. In Bitcoin, an epoch contains $L = 2016$ blocks. Thus, to support inclusion proofs for batches up to epoch size, the intermediary header Merkle tree must have a height of $\lceil \log_2 2016 \rceil = 11$. Consequently, a Merkle proof consists of eleven hashes in addition to the leaf header. To reduce the amount of data that needs to be sent to zkRelay contracts as well as the computational cost of on-chain Merkle proof validation, we validate the Merkle proof in a ZoKrates-based off-chain computation. The Merkle tree used for this intermediary header aggregation is independent of other Merkle trees used in the context of L_S or L_T. Hence, we can use any suitable collision-resistant hash function in its construction and are not constrained to SHA-256. In our implementation, we chose Pedersen hashes (see Section 8.3) for efficiency.

13.3.2 Performance Evaluation

In this section, we evaluate our zkRelay implementation with regards to off-chain processing times and memory consumption as well as on-chain processing costs. As intended with our design, the on-chain processing costs remain constant and do not depend on batch size. In contrast, the resource consumption of the off-chain step grows with the size of batches.

As a basis for our experiments, we created seven ZoKrates batch validation programs for different batch sizes. Hereby, the batch sizes were chosen so that they are fractions of epoch length L, i.e., so that $L \bmod N = 0$ (see Section 13.2.4). Like in our ZoKrates performance evaluation in Chapter 11, we distinguish between one-time and repeated steps.

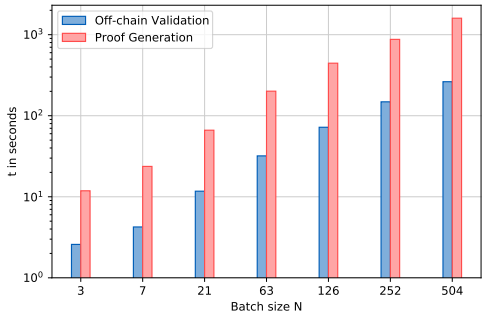
For our experiments, we used a Dell PowerEdge R540 server with an Intel Xeon Silver 4112 CPU@ 2.6 GHz, 128 GiB memory, and an SSD. We followed the experiment plan described in our performance evaluation in Section 11.1. The standard deviation between experiment runs was negligible, i.e., $\sigma < 1\%$, and is thus not reported explicitly.

Repeated Steps: Every batch validation in zkRelay involves running a ZoKrates batch validation program, generating a zk-SNARK proof that attests to the correctness of that program's execution. We provide an overview of the runtimes and memory consumption of both steps for different batch sizes in Figures 13.4a and 13.4b. Evidently, runtime and memory consumption depend on batch size (quasi-)linearly.

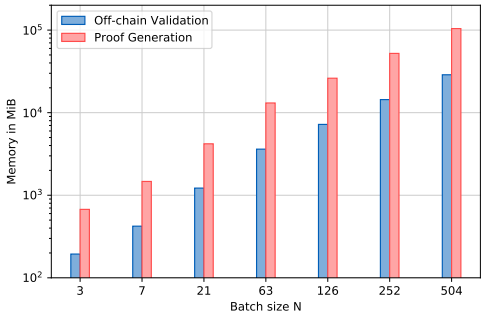
Proof generation represents the most memory-intensive step and thus implies an upper batch size that can be supported by given hardware. For example, proving an off-chain validation of a batch of 504 block headers requires ~104 GiB of RAM. With a memory consumption of ~13 GiB, batches of size 63 lend themselves to proving on standard consumer hardware.

For every off-chain batch validation, the zkRelay contract receives and validates a proof. In addition, it stores the last header of a batch. Crucially and by design, the cost of this on-chain processing is independent of batch size and always consumes 522,865 gas. Thereof, a cost of 351,226 gas applies for proof validation.

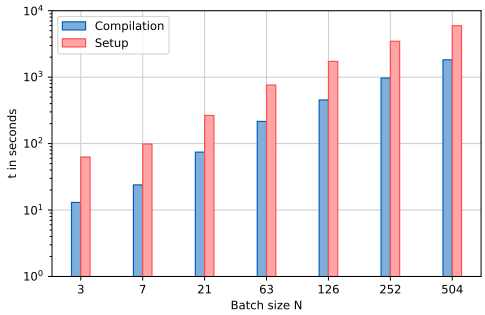
Validating a header in BTC Relay requires ~194,000 gas. Consequently, zkRelay is more effi-



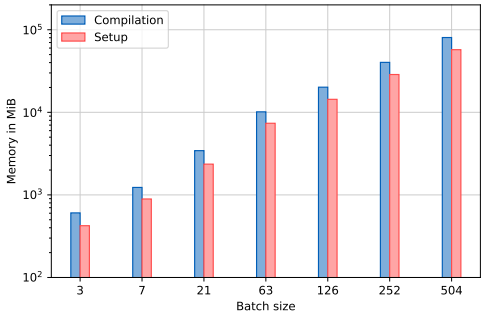
(a) Average runtime of repeated steps



(b) Average RAM required for repeated steps



(c) Average runtime of one-time steps



(d) Average RAM required for one-time steps

Figure 13.4: Runtime and Memory Consumption of Off-chain Validation and Proof Generation

cient than BTC Relay for batch sizes $N \geq 3$. Using BTC Relay, the validation of 504 headers, the largest batch size evaluated in this chapter, would consume an accumulated amount of 97,776,000 gas. Compared to zkRelay, this represents an 187× overhead.

One-time Steps: The compilation of ZoKrates programs, the related trusted setups, and the generation & deployment of zkRelay contracts is performed only once during the initialization of the zkRelay system. As shown in Figures 13.4c and 13.4d, the runtime and memory consumption of these steps again grow (quasi-)linearly with batch size. The ZoKrates program for the validation of a batch of 504 Bitcoin header compiles to 43,331,225 constraints. This translation consumes 80.50 GB of RAM. The one-time steps do imply minimum hardware requirements that have to be fulfilled to setup a zkRelay system. However, due to their one-time nature, users can outsource these steps to powerful, trusted hardware.

13.4 Related Work

In this section, we discuss work related to zkRelay. We summarize five relevant sidechaining proposals and compare them to our work. These proposals employ a variety of techniques to establish a trustless link between blockchains, ranging from direct on-chain verification to incentive-driven off-chain computations. We provide a comparison of the related proposals and their key charac-

teristics in Table 13.1.

Multiple sidechaining approaches have been proposed, performing the validation process directly in smart contracts, in off-chain computations, or by leveraging other proof constructions that are validated by the target ledger. In the following, we discuss three prominent mechanisms for sidechains and provide a brief overview of their characteristics in Table 13.1.

	BTC Relay	Dogethereum	NIPoPoW	ETH Relay	zkRelay
Source Ledger	Bitcoin	Dogecoin	No Implementation	Ethereum	Bitcoin
On-chain Computational Complexity	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$-$\mathcal{O}(n)$	$\mathcal{O}(1)$
Block Validation	On-Chain	Bulletpoof	NIPoPoW	On-Chain	zk-SNARK
Computation Validation	Smart Contract	Truebit	Smart Contract (Proof)	Smart Contract	Smart Contract (Proof)
Economic Rationality Assumptions	No	Yes	Yes	Yes	No
Fork required	No	No	Velvet Fork	No	No

Table 13.1: Comparison of different Sidechain Mechanisms

BTC Relay: BTC Relay [67] implements a one-way bridge between Bitcoin and Ethereum. Bitcoin headers are submitted sequentially to a BTC Relay smart contract where they are validated based on Bitcoin's consensus rules and stored in a header chain. Unlike zkRelay, BTC Relay does not support batch validation, leading to a significant cost overhead as demonstrated in Section 13.3.2. To incentivize the submission of Bitcoin headers and ensure the relay contract stays up-to-date, BTC Relay introduces a fee for SPV proofs that is redistributed to relayers to compensate for their gas costs [68]. Nevertheless, BTC Relay has stalled for over two years at the time of writing.

Dogethereum: With Dogethereum, Teutsch et al. proposed a design of an Ethereum-based relay for the Dogecoin blockchain [321]. To make ASIC-based mining more difficult, Dogecoin uses the memory-hard Scrypt key derivation function [8] in its PoW condition. As shown by Buterin, the direct on-chain verification of Dogecoin headers on Ethereum is impractical [76]. To address this challenge, Dogethereum combines Bulletproofs [69] with Truebit-based incentive-driven off-chain computations (see Section 5.2.3). First, the validity of a batch is proven in a Bulletproof. Since the validation of this proof on Ethereum is prohibitively expensive, it is published in the context of a Truebit verification game. Consequently, Dogethereum inherits Truebit's economic assumptions and security model.

NIPoPoWs: A blockchain relay based on Non-Interactive Proofs of Proof-of-Works (NIPoPoWs) has been proposed by Kiayias and Zindros [199]. NIPoPoWs are non-interactive proofs that can be used to convince clients of the occurrence of an event on a source blockchain [198]. They are succinct, i.e., proof size is polylogarithmic in the number of source blockchain blocks. In case the source blockchain does not natively support NIPoPoWs, it can be added through a comparatively uncontentious velvet fork [360]. Deployed PoW blockchains periodically adjust mining difficulty to achieve relatively constant block intervals. NIPoPoWs, however, require a constant difficulty assumption, which limits their applicability. Depending on the source blockchain's header structure, proofs can be concretely large (~15 KB in Ethereum [71]), rendering on-chain validation expensive. As shown by Gaži et al. [149], NIPoPoWs represent a promising primitive in the construction of relays for Proof-of-Stake (PoS) blockchains.

ETH Relay: With ETH Relay, Frauenthaler et al. proposed an optimistic validation-on-demand relay scheme that relies on economic incentives for the reporting of invalid block headers [140]. An ETH Relay contract accepts all submitted headers without validation but makes them inaccessible for SPV for a challenge period. During this period, so-called disputers can trigger on-chain validation to invalidate illegal headers, receive a reward, and trigger a punishment for the initial submitter. Thus, under the assumption of economically rational participants, on-chain validation becomes rare. This leads to significant cost savings compared to approaches with full validation by default. By design, the target ledger that contains a ETH Relay contract has to support the on-chain validation of a source ledger's header. Using ZoKrates-based off-chain header validation, as in zkRelay, ETH Relay could be augmented to even support source ledgers for which headers cannot be efficiently validated in an Ethereum smart contract.

13.5 Conclusion

In this chapter, we introduced zkRelay, an efficient blockchain relay facilitated by zk-SNARKs. Instead of storing and validating every block header of a source blockchain, a target blockchain's responsibility is reduced to validating a short proof generated in a verifiable off-chain computation that confirms the validity of a batch of source-chain headers. Unlike most other approaches, we do not rely on economic assumptions in our relay design.

With our ZoKrates-based zkRelay implementation, we demonstrated that ZoKrates can be used to address scalability concerns of blockchain-based applications: Concretely, we realized a chain-relay between Bitcoin and Ethereum that reduces the on-chain processing cost to a fraction of that of BTC Relay, the state-of-the-art solution representing our baseline.

CHAPTER 14

Anonymous Token Transfers

In this chapter, we provide an overview of Nightfall, a project that leverages ZoKrates to realize privacy-preserving transfers for fungible and non-fungible tokens on Ethereum.

Nightfall is independent work by the Ernst & Young Blockchain Research & Development team that heavily relies on ZoKrates for the implementation of its core components. As such, it demonstrates ZoKrates' viability as a practical framework for zk-SNARK-based verifiable off-chain computations.

We present the results in this chapter in the context of ZoKrates' evaluation and thus do not provide a complete and in-depth protocol description but refer the reader to the original specification [207] for that purpose. Nevertheless, we provide an overview of Nightfall's protocol design before discussing ZoKrates-related aspects of its implementation. The ZoKrates-based Nightfall implementation is available as open-source software [118].

14.1 Motivation

Smart contract–enabled blockchains like Ethereum allow the creation of user-defined digital assets, referred to as *tokens*, through smart contracts. To simplify the development of such tokens and to guarantee interoperability with wallet software and exchanges, standard interfaces have been defined: The ERC-20 [332] standard defines an interface for tokens that represent fungible assets, i.e., assets that are perfectly interchangeable. Each token has the same value. Non-fungible assets, in contrast, are not interchangeable due to their unique properties. Examples of such assets are certificates, collectibles, and non-standardized financial claims. Tokens representing such assets are commonly referred to as non-fungible tokens (NFTs). With ERC-721 [117], a standard interface for Ethereum-based NFTs was developed.

Like Ethereum's native token, user-defined tokens suffer from weak privacy guarantees. Ownership information is directly stored in token contracts and for each transfer, senders and receivers as well as the number of transferred tokens are visible to all network participants. Although

Ethereum addresses act as pseudonyms, they do not offer sufficient privacy protection since the analysis of associated blockchain transactions can reveal user identities [12, 239, 286, 291]. Building on Zerocoin [245], Zerocash [293] demonstrated how privacy-preserving cryptocurrency payments can be realized by leveraging zk-SNARKs. This proposal led to the advent of Zcash [361], a privacy-preserving cryptocurrency network. Zcash, however, only offers privacy protection for its protocol-native token and does not support user-defined assets.

To address this limitation, Nightfall extends the ideas of Zerocash [293] to support privacy-preserving transfers of Ethereum-based user-defined tokens. Concretely, it supports all token contracts that implement the ERC-20 or ERC-721 standard interfaces. Nightfall token transfers are always anonymous, i.e., hide sender and receiver. Depending on the token type, additional privacy guarantees are desirable and realized by the protocol: For ERC-20 tokens, the token amount is hidden. For ERC-721 tokens, which specific token is transferred remains secret.

14.2 Protocol Design

In this section, we provide an overview of the Nightfall protocol design and describe how it realizes privacy-preserving token transfers.

In regular Ethereum-based token transfers, a user signs a transaction with her private key and sends it to a token contract. This contract validates the signature and applies the transfer if the address corresponding to the private key has sufficient funds. In a privacy-preserving setting, the mapping between addresses and balances cannot be directly stored on-chain. Furthermore, sender and receiver addresses cannot be part of blockchain transactions.

To enable privacy-preserving token storage and transfers, Nightfall introduces a *shield contract*. This shield contract acts as a trustless escrow that receives tokens from users, tracks ownership, and performs transfers on users' behalf. Instead of directly storing ownership information for each token, it keeps a list of *token commitments*, i.e., cryptographic commitments that bind token ownership to owners' private keys without revealing that information on-chain. To transfer funds, users prove ownership of unspent token commitments to the shield contract, which then consumes these commitments and creates a new one bound to the receiver's private key. Crucially, consuming a token does not reveal the associated token commitment, i.e., which token was spent. Spent token commitments are nullified separately in an unlinkable way. The anonymity set thus consists of all past token commitments — spent or unspent.

The shield contract exposes three operations that are invoked by users: mint, transfer, and burn. We now describe these operations, the related Nightfall sub-protocols, and how they use zero-knowledge verifiable off-chain computations to realize privacy-preserving transfers in more detail. In our description, we follow Nightfall's naming conventions [207]. Due to the similarity of the protocols, we do not distinguish between hiding token amounts in the case of fungible tokens and hiding token identifiers in the case of NFTs. Instead, we focus on providing an intuition and describe a slightly simplified protocol for fungible tokens. For a complete formal protocol description, refer to the Nightfall documentation [207]. We provide an overview of the core components involved in these protocols in Figure 14.1.

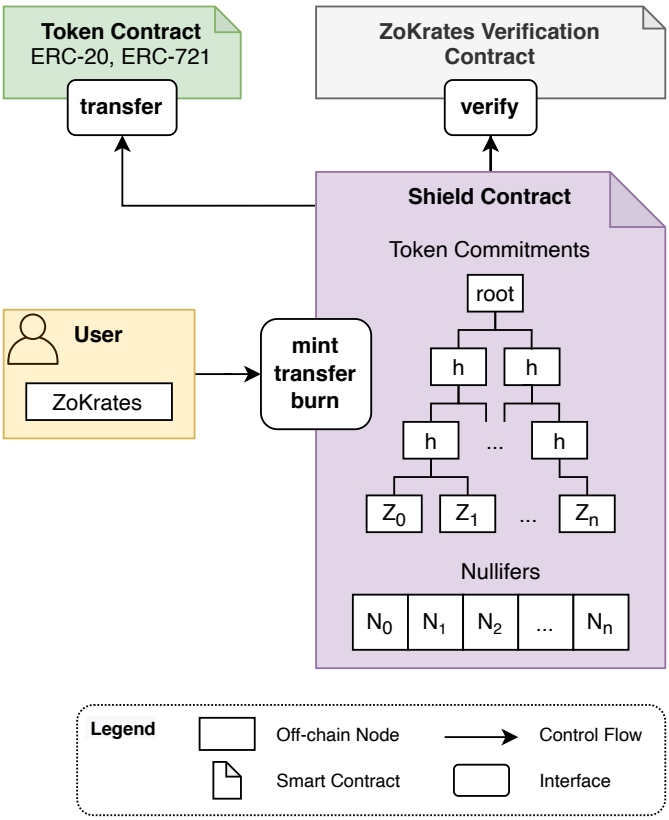


Figure 14.1: Overview of Nightfall's Core Components

14.2.1 Mint

In the minting step, tokens are transferred from a token owner to the shield contract, which acts as an escrow, in the token contract. The escrowed tokens are then linked with a token commitment that binds the deposited tokens to their initial owner without revealing that link on-chain. In other words, a standard token is converted into a shielded token represented on the blockchain through a token commitment.

Let h be a collision-resistant hash function. In Nightfall, h is instantiated with SHA-256 [134]. Let pk_o be the token owner's public key derived from her randomly sampled private key sk_o by applying this hash function, i.e., $pk_o = h(sk_o)$. The private key is used to prove ownership of a token commitment in later protocol steps.

First, a token owner decides on an amount of tokens α she wants to mint, i.e., convert into a token commitment. She then generates a random secret σ_o and computes a token commitment Z_o as follows:

$$Z_o = h(\alpha|pk_o|\sigma_o) \tag{14.1}$$

In addition to the token commitment Z_o, the shield contract must learn the token value of the

commitment to trigger the owner-to-escrow transfer. At the same time, the secret σ_o must not be revealed. The token owner achieves this through a verifiable off-chain computation: Using a ZoKrates program, the owner generates a zk-SNARK proof π that proves that Equation (14.1) holds for public inputs $i = (Z_o, \alpha)$ and private inputs $w = (pk_o, \sigma_o)$.

Then, the token owner invokes the shield contract's mint function with the resulting proof π, and public inputs $i = (Z_o, \alpha)$ as parameters. On successful verification, the shield contract stores the token commitment Z_o and invokes the token contract to transfer α tokens from the owners to itself. For efficiency, the shield contract uses a Merkle tree [241] as commitment tracking data structure. Note, that there is no anonymity at this point. A token commitment is linked to the owner's Ethereum address through the transfer in the token contract. The transfer amount α is also public.

14.2.2 Transfer

In the transfer step, a token owner — now a user with the knowledge to open a valid token commitment — sends a share of their tokens to another user.[1] Neither sender or recipient, nor the value of the transferred tokens is revealed in the process.

Let pk_r be the recipient's public key. We assume this public key to be known to the token owner initializing the transfer. Let $root$ be the current root hash of the token commitment tree and ψ_{Z_o} the corresponding Merkle inclusion proof for Z_o.

The token owner, acting as the sender, generates a random secret σ_t for the transfer. Then, she generates a new token commitment that links the tokens to the recipient:

$$Z_r = h(\alpha | pk_r | \sigma_t)$$

Additionally, she calculates a nullifier for the tokens commitment to be transferred:

$$N_o = h(\sigma_o | sk_o)$$

Nullifiers invalidate spent tokens. They are tracked by the shield contract to prevent double-spending without revealing the link between spent tokens and token commitments.

Subsequently, the owner uses a ZoKrates program to generate a zero-knowledge proof π that attests to the fact that the following assertions hold for public inputs $i = (Z_r, N_o, root)$ and private inputs $w = (\alpha, sk_p, pk_r, \sigma_o, \sigma_t, \psi_{Z_o})$:

$$\begin{aligned}
pk_o &= h(sk_o) && \text{(knowledge of private key)} \\
Z_o &= h(\alpha | pk_o | \sigma_o) && \text{(spending condition)} \\
Z_r &= h(\alpha | pk_r | \sigma_t) && \text{(preservation of value)} \\
N_o &= h(\sigma_o | sk_o) && \text{(nullifier correctness)} \\
is_merkle_proof(Z_o, root, \psi_{Z_o}) & && \text{(commitment exists in shield contract)}
\end{aligned}$$

[1] For simplicity, we describe a transfer of the full amount of tokens in a single token commitment. The real protocol is slightly more complex as it allows multiple input commitments as well as two output commitments where one acts as change.

To send the shielded tokens, the owner triggers the shield contract's transfer function with arguments $(i = (Z_r, N_o, root), \pi)$. On receipt, the shield contract verifies the proof π, confirms that $root$ has been a valid root of the token commitment Merkle tree at some point in time, and checks that N_o is not in the list of nullifiers. In case all checks succeed, it adds the new token commitment Z_r to the tree of token commitments and N_o to the list of nullifiers. Note, that neither information regarding the transferred token amount, nor the sender's or receiver's public keys are exposed to the blockchain in the process.

Finally, the owner sends Z_r, α, σ_t and Z_r's Merkle tree leaf index to the recipient through a secure channel. The recipient uses this information to validate the transfer and stores it for future transfers or burn transactions.

14.2.3 Burn

Shielded tokens can be converted back to regular tokens in an ERC-20 or ERC-721 token contract in the burn step. For that, on invalidation of a token commitment by a token owner, the shield contract transfers the associated escrowed tokens to the owner in the token contract. The burn operation reveals the number of tokens transferred as well as the token owner's Ethereum address in the process. The burnt token commitment is not revealed.

Let Z_o be the token commitment to be burnt. Further, let $root$ be the current root hash of the shield contract's token commitment Merkle tree and ψ_{Z_o} the corresponding Merkle inclusion proof for Z_o.

First, the token owner calculates a nullifier for the token commitment to be burnt:

$$N_o = h(\sigma_o | sk_o)$$

Then, she uses a ZoKrates program to generate a zk-SNARK proof π that attests to the fact that the following assertions hold for public inputs $i = (\alpha, N_o, root)$ and private inputs $w = (sk_o, \sigma_o, \psi_{Z_o})$:

$$\begin{aligned} pk_o &= h(sk_o) && \text{(knowledge of private key)} \\ Z_o &= h(\alpha | pk_o | \sigma_o) && \text{(spending condition)} \\ N_o &= h(\sigma_o | sk_o) && \text{(nullifier correctness)} \\ is\_merkle\_proof(Z_o, root, \psi_{Z_o}) && \text{(commitment exists in shield contract)} \end{aligned}$$

To burn the shielded tokens and obtain the corresponding token amount in the associated token contract, the owner invokes the shield contract's burn function with the proof π, the public inputs $i = (\alpha, N_o, root)$ and a target Ethereum address the escrowed tokens should be transferred to as arguments. The shield contract verifies π, asserts that $root$ has been a valid token commitment Merkle tree root in the past, and confirms that N_o is not in its nullifier list. On success, it adds N_o to the list of nullifiers and invokes the token contract to transfer token amount α from the escrow to the owner's target Ethereum address.

14.3 ZoKrates-based Implementation

In Nightfall, ZoKrates programs are used in the mint, transfer, and burn steps. Due to the slight differences between the protocols for fungible and non-fungible tokens, the ZoKrates programs for these steps also differ. Overall, the Nightfall implementation comprises the six ZoKrates programs displayed in Table 14.1 and available on GitHub [120]. We described their functionality as well as their public and private inputs in the previous section. In this section, we provide a brief overview of the properties of these programs as well as performance metrics for related on- and off-chain steps.

Table 14.1: Overview of ZoKrates Programs used in Nightfall Implementation

Protocol Step \ Token Type	ERC-20 (fungible)	ERC-721 (non-fungible)
mint	`ft-mint.zok`	`nft-mint.zok`
transfer	`ft-transfer.zok`	`nft-transfer.zok`
burn	`ft-burn.zok`	`nft-burn.zok`

14.3.1 Off-chain Steps

Table 14.2 shows key metrics for the off-chain steps associated with the ZoKrates program used in the Nightfall implementation. The programs are short (< 80 lines of code (LOC)); the LOC-metric refers to the core program stripped of comments and empty lines and excluding imported standard library and utility functions. Program execution and proving times are in the range of few to a hundred seconds and can thus be considered practical for many uses cases, especially those involving human users. The performance metrics were obtained using the same experiment setup and plan as in our performance evaluation (see Chapter 11). All cryptographic primitives used are included in the ZoKrates standard library.

Table 14.2: ZoKrates Program Metrics & Performance of Off-chain Steps

Program	Lines of Code	# Constraints	Witness Computation [s]	Proof Generation [s]
ft-mint	24	86 613	0.6	3.6
ft-transfer	73	2 357 944	9.5	99.6
ft-burn	38	1 150 474	4.7	48.9
nft-mint	35	114 723	0.7	4.0
nft-transfer	59	1 208 093	4.9	49.6
nft-burn	56	1 178 584	4.8	49.1

14.3.2 On-chain Steps

In Table 14.3, we provide performance metrics related to the on-chain processing steps that depend on ZoKrates-generated zk-SNARK proofs. The messages sent to the Nightfall shield contract by users are in the order of few hundred bytes. Thereof, the proof size is 256 bytes as the Nightfall implementation uses the GM17 verifiable computation scheme [166] which guarantees non-malleability (see Section 9.5). Note that the on-chain cost column reports the cost of all on-chain processing steps, not only the zk-SNARK verification in the ZoKrates verification contract.[2] To give a reasonable cost estimate in €, we use the same fixed gas and Ether prices as in our performance evaluation (given in Table 11.5).[3]

Table 14.3: ZoKrates-related Message Sizes and On-chain Processing Cost

Program	Message Size		On-chain Cost	
	Inputs [B]	Proof [B]	Gas [10^6]	€
ft-mint	128	256	1.8	2.67
ft-transfer	352	256	2.7	4.00
ft-burn	192	256	1.7	2.52
nft-mint	160	256	1.9	2.82
nft-transfer	224	256	2.1	3.11
nft-burn	224	256	1.8	2.67

14.4 Conclusion

In this chapter, we described Nightfall, an independently developed protocol for anonymous token transfers, and its implementation with ZoKrates. We provided a performance analysis of the off-chain execution and proving of ZoKrates programs as well as the on-chain processing of proofs generated therewith. The independent realization of Nightfall, a complex protocol, with the ZoKrates framework demonstrates ZoKrates' accessibility and viability. Our performance analysis underlines Nightfall's efficiency and hence shows that ZoKrates can be used for practical real-world applications.

In the future, the performance of ZoKrates-related protocol steps could be further improved by using crypto-primitives that are optimized for zk-SNARK-based off-chain computations and achieve lower constraint counts. Leveraging the ZoKrates standard library, for example, an alternative Nightfall version [119] uses MiMC [6] (see Section 8.3.4) instead of SHA-256 as collision-resistant hash function. This modification results in a $12\times$ speedup with regards to proof generation, but $3\times$ higher gas costs since no precompiles exist for MiMC on Ethereum at the time of writing. This limitation, however, is purely technical rather than conceptual.

[2] For the overall gas cost, we use the values reported in [207].

[3] Estimations based on network statistics observed on the Ethereum mainnet on 01.05.2020.

Part V

Conclusions

CHAPTER 15

Summary

Blockchains represent a promising category of distributed systems that allows a set of mutually distrusting parties to transact with each other without relying on a trusted third party. However, they suffer from a lack of scalability and in-built privacy protection mechanisms. In this thesis, we made four main contributions to address these challenges and to answer our research questions presented in Section 1.2.

In Part II, we introduced off-chaining as a fundamental approach to address privacy and scalability challenges in the context of blockchains and decentralized applications, thereby providing an answer to Research Question 1. After providing definitions, we proposed a set of off-chaining patterns that support concrete instantiations of the off-chaining idea to address recurring challenges in blockchain-based application design as our first contribution. As a second contribution, we explored the fundamental design space for off-chain computations, identified four fundamental categories of off-chain computation approaches based on shared assumptions and a common architecture, and provided a comprehensive and comparative overview of existing proposals from white and gray literature. We concluded that zero-knowledge verifiable off-chain computations are a particularly powerful approach to address scalability and privacy concerns in decentralized applications. At the same time, we found that developers are ill-equipped to leverage verifiable off-chain computations today.

To address this problem and to enable practical instantiations, we presented ZoKrates, the first comprehensive end-to-end framework for verifiable off-chain computations, in Part III of this thesis. It enables non-specialist developers to specify, integrate, and deploy zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK)-based off-chain computations. ZoKrates consists of a domain-specific language, a compiler, and tools for the generation of proofs and verification smart contracts for several state-of-the-art zk-SNARK verifiable computation schemes. It hides significant complexity inherent to zero-knowledge proofs and provides a more familiar and higher-level programming abstraction to developers, hence fostering adoption. As a concrete engineering tool for off-chain computations, ZoKrates provides an answer to Research Question 2.

In Part IV of this thesis, we applied ZoKrates to real-world use cases in the context of an extensive evaluation. We showed how ZoKrates can be leveraged to enhance the privacy or scalability properties of the three blockchain-based applications motivated in Chapter 1, thereby demonstrating its viability. Concretely, we designed and implemented a system for privacy-preserving energy sharing in a community of households as well as a scalable blockchain relay system an instantiation of which was shown to vastly improve the efficiency of a Bitcoin-Ethereum bridge. To underline ZoKrates' maturity, we described Nightfall, an independent ZoKrates-based realization of anonymous token transfers on Ethereum.

In summary, we advanced the state-of-the-art of privacy-aware and scalable decentralized application design in both theory and practice. We demonstrated the viability of off-chaining in general, and ZoKrates as a privacy and scalability engineering tool based on verifiable off-chain computations in particular.

CHAPTER 16

Outlook and Discussion

With our contributions presented in the previous chapters, we advanced the state of the art with regards to scalability and privacy engineering in the context of blockchain technologies. Nevertheless, there are several further research opportunities in the context of the problems addressed in this thesis. In this chapter, we briefly discuss how our line of work can be extended in the future and describe interesting on-going work in this area.

Off-chaining

In this thesis, we demonstrated how privacy and scalability concerns of blockchain-based applications can be addressed through off-chaining. For off-chaining approaches, we formulated the requirement that desirable properties gained from the adoption of blockchains should be compromised as little as possible in the process. In our patterns and off-chain computation approaches, we focused on preserving integrity and allowed liveness to be impaired. Solving the problem of data availability for off-chain data in a way that does not require trust in a third party remains an interesting open problem. Incentivized decentralized storage networks, e.g., Filecoin [278] and Swarm [324], represent a promising approach towards this goal for which the theoretical and practical boundaries have yet to be fully explored.

An interesting parallel line of work explores secure multi-party computation systems-as-a-service (MPSaaS) in the context of off-chaining [227, 322]. Although early-stage, MPC-based off-chain computation approaches have the potential to further improve on the privacy guarantees achievable through verifiable off-chain computations as no single party ever needs to know all private inputs to a computation.

Advances in Cryptography

Our work on off-chain computations presented in this thesis heavily relies on cryptographic schemes for non-interactive zero-knowledge proofs. In recent years, a plethora of such schemes have been published; we presented an overview in Chapter 2. While Groth16 [164] remains the

most efficient preprocessing zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) scheme in the context of our work, novel constructions that require weaker trust and security assumptions are developed at fast pace. Today, users face the dilemma of having to choose between performance, potentially even practicality, and weak assumptions. However, due to the high velocity that research progresses with, we expect a wider range of practical options to become available in the future, which allows more flexibility with regards to this tradeoff.

Ongoing research on circuit-friendly cryptographic primitives can contribute to significant performance improvement in verifiable off-chain computations. We already leverage such primitives, e.g., BabyJubJub [346] or MiMC [6], in ZoKrates and would directly benefit from novel constructions and improvements. In the context of decentralized applications, these primitives are likely to also be used on the blockchain and not exclusively in off-chain computations. Thus, an ideal primitive is efficient in circuits as well as in on-chain execution environments.

ZoKrates Framework Extensions

With our ZoKrates framework and its implementation, we showed that verifiable off-chain computations are usable in today's blockchain systems to increase scalability and improve the privacy of users. ZoKrates provides an expressive language and a modular design that facilitates the seamless adoption of innovations in a fast-moving environment. Still, there are many opportunities for future work, e.g., making the framework aware of recursive proof composition, utilizing specialized hardware or GPUs for proof generation, or natively supporting and coordinating different types of distributed setup protocols. Since ZoKrates programs have to be expected to manage or control large economic value in the context of off-chain computations, the correctness of their compilation and optimization is particularly important. Hence, exploring formal verification of key parts of the compilation and optimization steps may be worthwhile to minimize risks.

Applications

In an extensive evaluation, we demonstrated the viability of off-chaining in general and ZoKrates-based off-chain computations in particular for different applications. However, many other potential application domains yet remain to be explored. Expanding on our pattern collection, this process could be methodologically supported by characterizing blockchain-based applications that lend themselves to different off-chaining techniques.

APPENDIX A

ZoKrates Grammar

Listing A.1: ZoKrates Parsing Expression Grammar

```
/**
* ZoKrates Parsing Expression Grammar.
* Operator precedence rules handled in parsing process.
*/

file = { SOI ~ NEWLINE* ~ pragma? ~ NEWLINE* ~ import_directive* ~ NEWLINE* ~
     ty_struct_definition* ~ NEWLINE* ~ function_definition* ~ EOI }

pragma = { "#pragma" ~ "curve" ~ curve }
curve = @{ (ASCII_ALPHANUMERIC | "_") * }

import_directive = { main_import_directive | from_import_directive }
from_import_directive = { "from" ~ "\"" ~ import_source ~ "\"" ~ "import" ~
    identifier ~ ("as" ~ identifier)? ~ NEWLINE*}
main_import_directive = {"import" ~ "\"" ~ import_source ~ "\"" ~ ("as" ~
    identifier)? ~ NEWLINE+}
import_source = @{(!"\"" ~ ANY)*}
function_definition = {"def" ~ identifier ~ "(" ~ parameter_list ~ ")" ~
    return_types ~ ":" ~ NEWLINE* ~ statement* }
return_types = _{ ( "->" ~ ( "(" ~ type_list ~ ")" | ty ))? }

parameter_list = _{(parameter ~ ("," ~ parameter)*)?}
parameter = {vis? ~ ty ~ identifier}

// primitive types
ty_field = {"field"}
ty_bool = {"bool"}
ty_u8 = {"u8"}
ty_u32 = {"u32"}
ty_u16 = {"u16"}
ty_basic = { ty_field | ty_bool | ty_u8 | ty_u16 | ty_u32 }
// arrays
ty_array = { ty_basic_or_struct ~ ("[" ~ expression ~ "]")+ }
```

```
// structs
ty_struct = { identifier }

ty_basic_or_struct = { ty_basic | ty_struct }
ty = { ty_array | ty_basic | ty_struct }
type_list = _{(ty ~ ("," ~ ty)*)?}

// type definitions
ty_struct_definition = { "struct" ~ identifier ~ "{" ~ NEWLINE* ~
    struct_field_list ~ NEWLINE* ~ "}" ~ NEWLINE* }
struct_field_list = _{(struct_field ~ (NEWLINE+ ~ struct_field)*)? }
struct_field = { ty ~ identifier }

vis_private = {"private"}
vis_public = {"public"}
vis = { vis_private | vis_public }

// Statements
statement = { (return_statement // does not require subsequent newline
              | (iteration_statement
                | definition_statement
                | expression_statement
                ) ~ NEWLINE
            ) ~ NEWLINE* }

iteration_statement = { "for" ~ ty ~ identifier ~ "in" ~ expression ~ ".." ~
    expression ~ "do" ~ NEWLINE* ~ statement* ~ "endfor"}
return_statement = { "return" ~ expression_list}
definition_statement = { optionally_typed_assignee_list ~ "=" ~ expression }
expression_statement = {"assert" ~ "(" ~ expression ~ ")"}

optionally_typed_assignee_list = _{ optionally_typed_assignee ~ ("," ~
    optionally_typed_assignee)* }
optionally_typed_assignee = { (ty ~ assignee) | (assignee) }

// Expressions
expression_list = _{(expression ~ ("," ~ expression)*)?}

expression = { term ~ (op_binary ~ term)* }
term = { ("(" ~ expression ~ ")") | inline_struct_expression |
    conditional_expression | postfix_expression | primary_expression |
    inline_array_expression | array_initializer_expression | unary_expression
     }
spread = { "..." ~ expression }
range = { from_expression? ~ ".." ~ to_expression? }
from_expression = { expression }
to_expression = { expression }

conditional_expression = { "if" ~ expression ~ "then" ~ expression ~ "else" ~
     expression ~ "fi"}

postfix_expression = { identifier ~ access+ }
access = { array_access | call_access | member_access }
array_access = { "[" ~ range_or_expression ~ "]" }
```

```
call_access = { "(" ~ expression_list ~ ")" }
member_access = { "." ~ identifier }

primary_expression = { identifier
                     | constant
                     }

inline_struct_expression = { identifier ~ "{" ~ NEWLINE* ~
    inline_struct_member_list ~ NEWLINE* ~ "}" }
inline_struct_member_list = _{(inline_struct_member ~ ("," ~ NEWLINE* ~
    inline_struct_member)*)? ~ ","? }
inline_struct_member = { identifier ~ ":" ~ expression }

inline_array_expression = { "[" ~ NEWLINE* ~ inline_array_inner ~ NEWLINE* ~
    "]" }
inline_array_inner = _{(spread_or_expression ~ ("," ~ NEWLINE* ~
    spread_or_expression)*)?}
spread_or_expression = { spread | expression }
range_or_expression = { range | expression }
array_initializer_expression = { "[" ~ expression ~ ";" ~ constant ~ "]" }

unary_expression = { op_unary ~ term }

assignee = { identifier ~ assignee_access* }
assignee_access = { array_access | member_access }

// Identifiers
identifier = @{ ((!keyword ~ ASCII_ALPHA) | (keyword ~ (ASCII_ALPHANUMERIC |
    "_"))) ~ (ASCII_ALPHANUMERIC | "_")* }
constant = { hex_number | decimal_number | boolean_literal }
decimal_number = @{ "0" | ASCII_NONZERO_DIGIT ~ ASCII_DIGIT* }
boolean_literal = { "true" | "false" }
hex_number = _{ hex_number_32 | hex_number_16 | hex_number_8 }
hex_number_8 = @{ "0x" ~ ASCII_HEX_DIGIT{2} }
hex_number_16 = @{ "0x" ~ ASCII_HEX_DIGIT{4} }
hex_number_32 = @{ "0x" ~ ASCII_HEX_DIGIT{8} }

// Operators
op_or = @{"||"}
op_and = @{"&&"}
op_bit_xor = {"^"}
op_bit_and = {"&"}
op_bit_or = {"|"}
op_equal = @{"=="}
op_not_equal = @{"!="}
op_lt = {"<"}
op_lte = @{"<="}
op_gt = {">"}
op_gte = @{">="}
op_add = {"+"}
op_sub = {"-"}
op_mul = {"*"}
op_div = {"/"}
op_rem = {"%"}
```

```
op_pow = @{"**"}
op_not = {"!"}
op_left_shift = @{"<<"}
op_right_shift = @{">>"}
op_binary = _ { op_pow | op_or | op_and | op_bit_xor | op_bit_and | op_bit_or
     | op_left_shift | op_right_shift | op_equal | op_not_equal | op_lte |
    op_lt | op_gte | op_gt | op_add | op_sub | op_mul | op_div | op_rem }
op_unary = { op_not }

// Lexing Rules
WHITESPACE = _{ " " | "\t" | "\\" ~ NEWLINE}
COMMENT = _{ ("/*" ~ (!"*/" ~ ANY)* ~ "*/") | ("//" ~ (!NEWLINE ~ ANY)*) }

keyword = @{"assert"|"as"|"bool"|"byte"|"def"|"do"|"else"|"endfor"|"export"|"
    false"|"field"|"for"|"if"|"then"|"fi"|"import"|"from"|
             "in"|"private"|"public"|"return"|"struct"|"true"|"u8"|"u16"|"u32"
             }
```

APPENDIX B

Selected Supervised Theses

Over the period of time we worked towards this thesis, we supervised several bachelor's and master's theses. A first feasibility study of a DSL for verifiable off-chain computations was conducted by Dennis Kuhnert in [212]. The positive results motivated the development of ZoKrates. Several other theses in the wider context of off-chaining shaped our way of thinking about the field through their contributions and related discussions: Steffen Härtlein developed a blockchain-coordinated distributed setup procedure for zk-SNARKs [170]. Patrick Friedrich demonstrated the viability and efficiency of ZoKrates by implementing a proof-of-storage protocol [142] as used in Filecoin [278]. Dennis Westphal combined several off-chaining patterns to realize a privacy-preserving apartment rental dApp [344]. With Nightlines, Dong-Ha Kim developed an extension of Nightfall (see Chapter 14) to realize privacy-preserving IOU networks [201].

List of Publications

J. Eberhardt, D. Ernst, and D. Bermbach. “SMAC: State Management for Geo-Distributed Containers”. In: *Proceedings of the 2nd International Workshop on Container Technologies and Container Clouds (WoC)*. IEEE. 2016.

J. Eberhardt and J. Heiss. “Off-chaining Models and Approaches to Off-chain Computations”. In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 2018, pp. 7–12.

J. Eberhardt, M. Peise, D.-H. Kim, and S. Tai. “Privacy-Preserving Netting in Local Energy Grids”. In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 2020, pp. 1–9.

J. Eberhardt and S. Tai. “On or Off the Blockchain? Insights on Off-Chaining Computation and Data”. In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 3–15.

J. Eberhardt and S. Tai. “ZoKrates - Scalable Privacy-Preserving Off-Chain Computations”. In: *IEEE International Conference on Blockchain*. (Best Paper Award). IEEE. 2018, pp. 1084–1091.

D. Bermbach and J. Eberhardt. “Audio-Visual Cues for Cloud Service Monitoring”. In: *Proceedings of the International Conference on Cloud Computing and Services Science*. ScitePress, 2017.

D. Bermbach and J. Eberhardt. “Towards Audio-Visual Cues for Cloud Infrastructure Monitoring”. In: *IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2016, pp. 218–219.

D. Bermbach, S. Müller, J. Eberhardt, and S. Tai. “Informed Schema Design for Column Store-Based Database Services”. In: *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2015, pp. 163–172.

A. Busse, J. Eberhardt, S. Frost, D.-H. Kim, T. Weilbier, L. Renner, M. Roth, and S. Tai. “A Response to the CITES Blockchain Challenge: Incremental and Integrative PoA-based Permit Exchange”. In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 2019.

A. Busse, J. Eberhardt, and S. Tai. “EVM-Perf: High-Precision EVM Performance Analysis”. In: *IEEE International Conference on Blockchain and Cryptocurrency 2021*. IEEE, 2021.

J. Heiss, J. Eberhardt, and S. Tai. “From Oracles to Trustworthy Data On-chaining Systems”. In: *IEEE International Conference on Blockchain*. IEEE. 2019.

J. Heiss, M.-R. Ulbricht, and J. Eberhardt. “Put Your Money Where Your Mouth Is – Towards Blockchain-based Consent Violation Detection”. In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency 2020*. IEEE, 2020.

G. Katsaros, M. Menzel, A. Lenk, J. R. Revelant, R. Skipp, and J. Eberhardt. “Cloud Application Portability with TOSCA, Chef and Openstack”. In: *IEEE International Conference on Cloud Engineering*. IEEE, 2014, pp. 295–302.

M. Klems, J. Eberhardt, S. Tai, S. Härtlein, S. Buchholz, and A. Tidjani. “Trustless intermediation in blockchain-based decentralized service marketplaces”. In: *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 731–739.

S. Tai, J. Eberhardt, and M. Klems. “Not ACID, not BASE, but SALT – A Transaction Processing Perspective on Blockchains”. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. ScitePress, 2017, pp. 755–764.

M. Westerkamp and J. Eberhardt. “zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays”. In: *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*. IEEE, 2020, pp. 378–386.

Bibliography

[1] I. Abraham, B. Pinkas, and A. Yanai. “Blinder–Scalable, Robust Anonymous Committed Broadcast”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2020, pp. 1233–1252.

[2] G. Ács and C. Castelluccia. “I Have a DREAM! (DiffeRentially privatE smArt Metering)”. In: *Information Hiding*. Ed. by T. Filler, T. Pevný, S. Craver, and A. Ker. Springer, 2011, pp. 118–132. DOI: 10.1007/978-3-642-24178-9_9.

[3] A. Ahadipour, M. Mohammadi, and A. Keshavarz-Haddad. “Statistical-Based Privacy-Preserving Scheme with Malicious Consumers Identification for Smart Grid”. In: (2019). arXiv: 1904.06576.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] N. Z. Aitzhan and D. Svetinovic. “Security and Privacy in Decentralized Energy Trading Through Multi-Signatures, Blockchain and Anonymous Messaging Streams”. In: *IEEE Transactions on Dependable and Secure Computing* 15 (2018), pp. 840–852. DOI: 10.1109/TDSC.2016.2616861.

[6] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen. “MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 191–219.

[7] T. Alves and D. Felton. “TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems”. In: *ARM IQ* 3.4 (2014).

[8] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro. “Scrypt Is Maximally Memory-Hard”. In: *Advances in Cryptology – EUROCRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer, 2017, pp. 33–62.

[9] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. “Ligero: Lightweight sublinear arguments without a trusted setup”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2087–2104.

[10] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. “Innovative technology for CPU based attestation and sealing”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Vol. 13. Citeseer, 2013, p. 7.

[11] E. Androulaki, S. Cocco, and C. Ferris. *Private and confidential transactions with Hyperledger Fabric*. 2018. URL: https://developer.ibm.com/tutorials/cl-

blockchain-private-confidential-transactions-hyperledger-fabric-zero-knowledge-proof/ (visited on 07. 01. 2021).

[12] E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun. "Evaluating user privacy in bitcoin". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 34–51.

[13] E. Androulaki et al. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. ACM, 2018. DOI: 10.1145/3190508.3190538.

[14] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. "Secure multiparty computations on bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 443–458.

[15] J. Angelis and E. R. da Silva. "Blockchain adoption: A value driver perspective". In: *Business Horizons* 62.3 (2019), pp. 307–314.

[16] K. Arnold, J. Gosling, D. Holmes, and D. Holmes. *The Java programming language*. Vol. 2. Addison-Wesley Reading, 2000.

[17] S. Arora and B. Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[18] M. R. Asghar, G. Dán, D. Miorandi, and I. Chlamtac. "Smart Meter Data Privacy: A Survey". In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2820–2835. DOI: 10.1109/COMST.2017.2720195.

[19] Y. Assia, V. Buterin, M. R. liorhakiLior Hakim, and R. Lev. *Colored coins whitepaper*. 2012. URL: https://docs.google.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit (visited on 07. 01. 2021).

[20] G. Avarikioti, E. K. Kogias, R. Wattenhofer, and D. Zindros. *Brick: Asynchronous Payment Channels*. 2019. arXiv: 1905.11360.

[21] L. Babai. "Trading group theory for randomness". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. ACM, 1985, pp. 421–429.

[22] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille. *Enabling blockchain innovations with pegged sidechains*. 2014. URL: https://www.blockstream.com/sidechains.pdf (visited on 25. 06. 2020).

[23] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. "SoK: Consensus in the Age of Blockchains". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM, 2019, pp. 183–198.

[24] B. Barak. *An Intensive Introduction to Cryptography*. 2020. URL: https://intensecrypto.org/ (visited on 19. 11. 2020).

[25] B. Barak. *Introduction to Theoretical Computer Science*. 2020. URL: https://introtcs.org/ (visited on 30. 01. 2020).

[26] E. Barker, L. Chen, S. Keller, A. Roginsky, A. Vassilev, and R. Davis. *Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography*. 2017.

[27] P. S. Barreto and M. Naehrig. "Pairing-friendly elliptic curves of prime order". In: *International Workshop on Selected Areas in Cryptography*. Springer, 2005, pp. 319–331.

[28] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[29] C. Baum, I. Damgård, and C. Orlandi. “Publicly auditable secure multi-party computation”. In: *International Conference on Security and Cryptography for Networks*. Springer, 2014, pp. 175–196.

[30] M. Baza, M. Nabil, N. Lasla, K. Fidan, M. Mahmoud, and M. Abdallah. “Blockchain-based firmware update scheme tailored for autonomous vehicles”. In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–7.

[31] BDEW. *Blockchain in the Energy Sector - The Potential for Energy Providers*. 2018. URL: https://www.bdew.de/media/documents/Studie-Blockchain-englische-Fassung-Dez.2018.pdf (visited on 07. 01. 2021).

[32] M. Bellare and P. Rogaway. “Random oracles are practical: A paradigm for designing efficient protocols”. In: *Proceedings of the 1st ACM conference on Computer and Communications Security*. ACM, 1993, pp. 62–73.

[33] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*. 2018. Cryptology ePrint Archive: 2018.046.

[34] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. “Fast reductions from RAMs to delegatable succinct constraint satisfaction problems”. In: *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*. ACM, 2013, pp. 401–414.

[35] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. “SNARKs for C: Verifying program executions succinctly and in zero knowledge”. In: *Advances in Cryptology — CRYPTO 2013*. Springer, 2013, pp. 90–108.

[36] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. “Secure sampling of public parameters for succinct zero knowledge proofs”. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 287–304.

[37] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. “Aurora: Transparent succinct arguments for R1CS”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 103–128.

[38] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture.” In: *USENIX Security Symposium (USENIX Security)*. USENIX. 2014, pp. 781–796.

[39] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. “Robust PCPs of proximity, shorter PCPs, and applications to coding”. In: *SIAM Journal on Computing* 36.4 (2006), pp. 889–974.

[40] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. arXiv: 1407.3561.

[41] I. Bentov, R. Kumaresan, and A. Miller. “Instantaneous decentralized poker”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 410–440.

[42] J. Bergquist, A. Laszka, M. Sturm, and A. Dubey. “On the design of communication and transaction anonymity in blockchain-based transactive microgrids”. In: *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 2017, pp. 1–6. DOI: 10.1145/3152824.3152827.

[43] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. “Twisted edwards curves”. In: *International Conference on Cryptology in Africa*. Springer, 2008, pp. 389–405.

[44] D. J. Bernstein, T. Lange, et al. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. 2013. URL: http://safecurves.cr.yp.to (visited on 10. 01. 2021).

[45] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. "On the indifferentiability of the sponge construction". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2008, pp. 181–197.

[46] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. "From Extractable Collision Resistance to Succinct Non-interactive Arguments of Knowledge, and Back Again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 326–349. DOI: 10.1145/2090236.2090263.

[47] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky. "Succinct non-interactive arguments via linear interactive proofs". In: *Theory of Cryptography Conference*. Springer, 2013, pp. 315–333.

[48] Bitcoin Wiki. URL: https://en.bitcoin.it/wiki/Scalability (visited on 30. 09. 2020).

[49] *Bitcoin Wiki Protocol Documentation*. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 04. 11. 2020).

[50] "Blockchain technology in the energy sector: A systematic review of challenges and opportunities". In: *Renewable and Sustainable Energy Reviews* 100 (2019), pp. 143–174. DOI: 10.1016/j.rser.2018.10.014.

[51] M. Blum. "Coin flipping by telephone a protocol for solving impossible problems". In: *ACM SIGACT News* 15.1 (1983), pp. 23–27.

[52] M. Blum, P. Feldman, and S. Micali. "Non-interactive Zero-knowledge and Its Applications". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. ACM, 1988, pp. 103–112. DOI: 10.1145/62212.62222.

[53] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro. *Coda: Decentralized Cryptocurrency at Scale*. 2020. Cryptology ePrint Archive: 2020.352.

[54] S. Bowe. *bellman - zkSNARK library*. URL: https://github.com/zkcrypto/bellman (visited on 06. 11. 2019).

[55] S. Bowe. *Powers of Tau MPC Rust implementation*. URL: https://github.com/ebfull/powersoftau (visited on 20. 04. 2020).

[56] S. Bowe. *zCash jubjub prototype*. URL: https://github.com/Electric-Coin-Company/jubjub-prototype (visited on 09. 09. 2019).

[57] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. "Zexe: Enabling decentralized private computation". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 947–964.

[58] S. Bowe, A. Gabizon, and M. D. Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. 2017. Cryptology ePrint Archive: 2017.602.

[59] S. Bowe, A. Gabizon, and I. Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. 2017. Cryptology ePrint Archive: 2017.1050.

[60] Brainbot Labs. *Raiden Network*. 2017. URL: http://raiden.network/ (visited on 07. 01. 2010).

[61] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. "Characterizing Efficiency Optimizations in Solidity Smart Contracts". In: *2020 IEEE International Conference on*

Blockchain (Blockchain). 2020, pp. 281–290. DOI: 10.1109/Blockchain50366.2020.00042.

[62] G. Brassard, D. Chaum, and C. Crépeau. "Minimum disclosure proofs of knowledge". In: *Journal of computer and system sciences* 37.2 (1988), pp. 156–189.

[63] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. *Software Grand Exposure: SGX Cache Attacks Are Practical*. 2017. arXiv: 1702.07521.

[64] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. "Verifying computations with state". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 341–357.

[65] C. Braun and T. Käfer. "Verifying the Integrity of Hyperlinked Information Using Linked Data and Smart Contracts". In: *Semantic Systems. The Power of AI and Knowledge Graphs*. Springer, 2019, pp. 376–390.

[66] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. *Corda: an introduction*. 2016. URL: https://www.r3.com/wp-content/uploads/2019/06/corda-platform-whitepaper.pdf (visited on 12. 11. 2020).

[67] *BTC Relay*. URL: http://btcrelay.org (visited on 18. 08. 2020).

[68] *BTC Relay Incentives for Relayers*. URL: https://github.com/ethereum/btcrelay/tree/master#incentives-for-relayers (visited on 18. 10. 2020).

[69] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. "Bulletproofs: Short proofs for confidential transactions and more". In: *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[70] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent snarks from dark compilers". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 677–706.

[71] B. Bünz, L. Kiffer, L. Luu, and M. Zamani. "Flyclient: Super-light clients for cryptocurrencies". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 928–946.

[72] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. "HyCC: Compilation of hybrid protocols for practical secure computation". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 847–861.

[73] V. Buterin. *Chain Interoperability*. 2016. URL: https://www.r3.com/wp-content/uploads/2017/06/chain_interoperability_r3.pdf (visited on 15. 02. 2020).

[74] V. Buterin. *Ethereum: A next-generation smart contract and decentralized application platform*. 2014. URL: https://github.com/ethereum/wiki/wiki/%5C%5BEnglish%5C%5D-White-Paper (visited on 15. 02. 2020).

[75] V. Buterin. *On public and private blockchains*. 2015. URL: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/ (visited on 12. 11. 2020).

[76] V. Buterin. *Scrypt in EVM for use in verifying dogecoin block headers*. URL: https://www.reddit.com/r/dogecoin/comments/3xc0co/dogeethereum_twoway_peg_i_wrote_up_an/ (visited on 15. 06. 2020).

[77] Carlos Pérez Baró. *Ethereum Stackexchange - How many transactions can the Ethereum network handle?* URL: https://ethereum.stackexchange.com/questions/49484/how-many-transactions-per-second-can-ethereum-currently-handle-what-changes-wil (visited on 05. 11. 2019).

[78] M. Chaouch. "Clustering-based improvement of nonparametric functional time series forecasting: Application to intra-day household-level load curves". In: *IEEE Transactions on Smart Grid* 5.1 (2013), pp. 411–419.

[79] J. M. Chase. *Quorum Whitepaper, Version 0.1.* 2016. URL: https://github.com/jpmorganchase/quorum-docs/blob/master/Quorum%20Whitepaper%20v0.1.pdf (visited on 12. 11. 2020).

[80] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn. "Malleable proof systems and applications". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2012, pp. 281–300.

[81] D. L. Chaum. "Untraceable electronic mail, return addresses, and digital pseudonyms". In: *Communications of the ACM* 24.2 (1981), pp. 84–90.

[82] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[83] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. *Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution*. 2018. arXiv: 1804.05141.

[84] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. "Marlin: Preprocessing zksnarks with universal and updatable srs". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 738–768.

[85] A. Chiesa, D. Ojha, and N. Spooner. "Fractal: Post-quantum and transparent recursive proofs from holography". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 769–793.

[86] K.-M. Chung, Y. Kalai, and S. Vadhan. "Improved delegation of computation using fully homomorphic encryption". In: *Annual Cryptology Conference*. Springer, 2010, pp. 483–501.

[87] R. Cohen and Y. Lindell. "Fairness versus guaranteed output delivery in secure multiparty computation". In: *Journal of Cryptology* 30.4 (2017), pp. 1157–1186.

[88] J. Coleman. *State Channels*. 2015. URL: https://www.jeffcoleman.ca/statechannels/ (visited on 25. 06. 2020).

[89] S. A. Cook and R. A. Reckhow. "Time bounded random access machines". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375.

[90] V. Costan and S. Devadas. "Intel SGX Explained." In: (2016), pp. 1–118. Cryptology ePrint Archive: 2016.86.

[91] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. "Geppetto: Versatile verifiable computation". In: *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, pp. 253–270.

[92] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, et al. "On scaling decentralized blockchains". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 106–125.

[93] E. Dall'Anese, H. Zhu, and G. B. Giannakis. "Distributed Optimal Power Flow for Smart Microgrids". In: *IEEE Transactions on Smart Grid* 4.3 (2013), pp. 1464–1475. DOI: 10.1109/TSG.2013.2248175.

[94] I. Damgård. "Efficient concurrent zero-knowledge in the auxiliary string model". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2000, pp. 418–430.

[95] G. Danezis, C. Fournet, J. Groth, and M. Kohlweiss. "Square span programs with applications to succinct NIZK arguments". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2014, pp. 532–550.

[96] dApp.com - Decentralized Application Registry. URL: https://www.dapp.com/ (visited on 10. 10. 2020).

[97] DappRadar - Decentralized Application Registry. URL: https://dappradar.com / (visited on 10. 10. 2020).

[98] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi. "FastKitten: Practical Smart Contracts on Bitcoin". In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX, 2019, pp. 801–818.

[99] DENA. *Blockchain in the integrated energy transition*. 2019. URL: https://www.dena.de/fileadmin/dena/Publikationen/PDFs/2019/dena-Studie_Blockchain_Integrierte_Energiewende_EN2.pdf (visited on 07. 01. 2021).

[100] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[101] C. Diligence. *Smart Contract Best Practices — Pull over Push*. URL: https://consensys.github.io/smart-contract-best-practices/recommendations/ (visited on 11. 01. 2021).

[102] T. Dimitriou and G. Karame. "Privacy-friendly tasking and trading of energy in smart grids". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 652–659.

[103] I. Dinur. "The PCP theorem by gap amplification". In: *Journal of the ACM (JACM)* 54.3 (2007).

[104] K. Doka, T. Bakogiannis, I. Mytilinis, and G. Goumas. "CloudAgora: Democratizing the Cloud". In: *Proceedings of the 2nd International Conference on Blockchain*. Springer, 2019, pp. 142–156.

[105] C. Donnelly and R. Stallman. *Bison. The Yacc-compatible Parser Generator, Version 3*. 2007.

[106] A. Dorri, F. Luo, S. S. Kanhere, R. Jurdak, and Z. Y. Dong. *SPB: A Secure Private Blockchain-based Solution for Energy Trading*. 2018. arXiv: 1807.10897 [cs.CR].

[107] J. R. Douceur. "The Sybil Attack". In: *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 251–260.

[108] P. Dunphy and F. A. Petitcolas. "A first look at identity management schemes on the blockchain". In: *IEEE Security & Privacy* 16.4 (2018), pp. 20–29.

[109] J. B. Earp, A. I. Anton, L. Aiman-Smith, and W. H. Stufflebeam. "Examining Internet privacy policies within the context of user privacy values". In: *IEEE Transactions on*

Engineering Management 52.2 (2005), pp. 227–237. DOI: 10.1109/TEM.2005.844927.

[110] J. Eberhardt and J. Heiss. "Off-chaining Models and Approaches to Off-chain Computations". In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. ACM, 2018, pp. 7–12. DOI: 10.1145/3284764.3284766.

[111] J. Eberhardt, M. Peise, D.-H. Kim, and S. Tai. "Privacy-Preserving Netting in Local Energy Grids". In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency 2020*. IEEE, 2020. DOI: 10.1109/ICBC48266.2020.9169440.

[112] J. Eberhardt and S. Tai. "On or Off the Blockchain? Insights on Off-Chaining Computation and Data". In: *Proceedings of the European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 3–15. DOI: 10.1007/978-3-319-67262-5_1.

[113] J. Eberhardt and S. Tai. "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations". In: *IEEE International Conference on Blockchain*. IEEE, 2018. DOI: 10.1109/Cybermatics_2018.2018.00199.

[114] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 2019. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm (visited on 07. 01. 2021).

[115] C. Efthymiou and G. Kalogridis. "Smart Grid Privacy via Anonymization of Smart Metering Data". In: *2010 First IEEE International Conference on Smart Grid Communications*. IEEE, 2010, pp. 238–243. DOI: 10.1109/SMARTGRID.2010.5622050.

[116] Enigma MPC, Inc. *Enigma Protocol Documentation*. URL: https://www.enigma.co/protocol/index.html (visited on 22. 10. 2020).

[117] W. Entriken, D. Shirley, J. Evans, and N. Sachs. *ERC-721 Non-Fungible Token Standard*. 2018. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md (visited on 16. 11. 2020).

[118] Ernst & Young Blockchain R&D. *Nightfall Implementation*. URL: https://github.com/EYBlockchain/nightfall/ (visited on 20. 10. 2020).

[119] Ernst & Young Blockchain R&D. *Nightlite Documentation*. URL: https://github.com/EYBlockchain/nightlite/blob/b5437ef0ff71a99fdd58ef531fb6ac41062db630/README.md (visited on 10. 11. 2020).

[120] Ernst & Young Blockchain R&D. *Source Code of ZoKrates Programs used in Nightfall*. URL: https://github.com/EYBlockchain/nightlite/tree/b5437ef0ff71a99fdd58ef531fb6ac41062db630/setup/gm17 (visited on 10. 11. 2020).

[121] *Ethereum as a Service on Microsoft Azure*. URL: https://azure.microsoft.com/de-de/blog/ethereum-blockchain-as-a-service-now-on-azure/ (visited on 30. 09. 2020).

[122] *Ethereum Chess Proof-of-Concept Implementation*. URL: https://github.com/ise-ethereum/on-chain-chess (visited on 15. 06. 2020).

[123] Ethereum Foundation. *Ethereum Roadmap: ZK-Rollups*. URL: https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/ (visited on 03. 11. 2020).

[124] Ethereum Foundation. *LLL Language Documentation*. 2017. URL: https://lll-docs.readthedocs.io/en/latest/ (visited on 07. 10. 2020).
[125] Ethereum Foundation. *Remix Web IDE for Ethereum Smart Contract Development*. URL: https://remix.ethereum.org/ (visited on 21. 04. 2020).
[126] Ethereum Foundation. *Solidity Language Documentation, v0.6.3*. 2020. URL: https://solidity.readthedocs.io/en/v0.6.3/ (visited on 22. 02. 2020).
[127] Ethereum Foundation. *Vyper Language Documentation, v0.2.5*. 2020. URL: https://vyper.readthedocs.io/en/v0.2.5/ (visited on 07. 10. 2020).
[128] Ethereum Foundation. *web3.js - Ethereum Node JavaScript API*. URL: https://github.com/ethereum/web3.js/ (visited on 15. 02. 2020).
[129] Ethereum Foundation and Protocol Labs. *MiMC Hash Challenge*. 2020. URL: https://mimchash.org/ (visited on 29. 03. 2020).
[130] *Ethereum Improvement Proposals (EIPs)*. URL: https://github.com/ethereum/EIPs (visited on 07. 01. 2021).
[131] *Ethereum Whisper Protocol v5*. URL: https://geth.ethereum.org/docs/whisper/whisper-overview (visited on 10. 01. 2021).
[132] EY Blockchain. URL: https://www.ey.com/en_gl/blockchain (visited on 17. 11. 2019).
[133] A. Fiat and A. Shamir. "How to prove yourself: Practical solutions to identification and signature problems". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 186–194.
[134] P. FIPS. "180-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION". In: *Secure Hash Standard (SHS), National Institute of Standards and Technology* (2012).
[135] K. Floersch. "Ethereum Smart Contracts in L2: Optimistic Rollup". In: *https://medium.com/plasma-group/ethereum-smart-contracts-in-l2-optimistic-rollup-2c1cef2ec537* (2019).
[136] B. Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2004, pp. 111–122. DOI: 10.1145/964001.964011.
[137] Fortanix. *Fortanix Rust Enclave Development Platform*. URL: https://edp.fortanix.com/ (visited on 10. 07. 2020).
[138] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. "ZQL: A Compiler for Privacy-Preserving Data Processing". In: *22nd USENIX Security Symposium (USENIX Security)*. USENIX, 2013, pp. 163–178.
[139] S. J. Fowler. *Production-ready microservices: Building standardized systems across an engineering organization*. O'Reilly Media, Inc., 2016.
[140] P. Frauenthaler, M. Sigwart, C. Spanring, M. Sober, and S. Schulte. "ETH Relay: A Cost-efficient Relay for Ethereum-based Blockchains". In: *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE. 2020, pp. 204–213.
[141] M. Fredrikson and B. Livshits. "ZØ: An optimizing distributing zero-knowledge compiler". In: *23rd USENIX Security Symposium (USENIX Security)*. USENIX, 2014, pp. 909–924.

[142] P. Friedrich. “Applied zkSNARKs: ZoKrates-based Implementation of Proof-of-Storage”. MA thesis. Technische Universität Berlin, 2019.

[143] D. Gabay, K. Akkaya, and M. Cebe. “A Privacy Framework for Charging Connected Electric Vehicles Using Blockchain and Zero Knowledge Proofs”. In: *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*. IEEE, 2019, pp. 66–73.

[144] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. 2019. Cryptology ePrint Archive: 2019.953.

[145] K. Gai, Y. Wu, L. Zhu, M. Qiu, and M. Shen. “Privacy-Preserving Energy Trading Using Consortium Blockchain in Smart Grid”. In: *IEEE Transactions on Industrial Informatics* 15.6 (2019), pp. 3548–3558. DOI: 10.1109/TII.2019.2893433.

[146] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994.

[147] J. von zur Gathen and G. Seroussi. “Boolean circuits versus arithmetic circuits”. In: *Information and Computation* 91.1 (1991), pp. 142–154.

[148] A. Gautier, J. Jacqmin, and J.-C. Poudou. “The prosumers and the grid”. In: *Journal of Regulatory Economics* 53.1 (2018), pp. 100–126.

[149] P. Gaži, A. Kiayias, and D. Zindros. “Proof-of-stake sidechains”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 139–156.

[150] R. Gennaro, C. Gentry, and B. Parno. “Non-interactive verifiable computing: Outsourcing computation to untrusted workers”. In: *Annual Cryptology Conference*. Springer, 2010, pp. 465–482.

[151] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. *Quadratic Span Programs and Succinct NIZKs without PCPs*. 2012. Cryptology ePrint Archive: 2012.215.

[152] C. Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, 2009.

[153] C. Gentry and D. Wichs. “Separating succinct non-interactive arguments from all falsifiable assumptions”. In: *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 2011, pp. 99–108.

[154] C. Goebel, H.-A. Jacobsen, V. del Razo, C. Doblander, J. Rivera, J. Ilg, C. Flath, H. Schmeck, C. Weinhardt, D. Pathmaperuma, et al. “Energy informatics”. In: *Business & Information Systems Engineering* 6.1 (2014), pp. 25–31.

[155] O. Goldreich, S. Micali, and A. Wigderson. “How to Play ANY Mental Game”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. ACM, 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[156] O. Goldreich, S. Micali, and A. Wigderson. “How to prove all NP statements in zero-knowledge and a methodology of cryptographic protocol design”. In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 171–185.

[157] S. Goldwasser, S. Micali, and C. Rackoff. “The knowledge complexity of interactive proof systems”. In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[158] L. Goodman. *Tezos: A self-amending crypto-ledger position paper*. URL: https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf (visited on 25. 08. 2020).

[159] A. Goranović, M. Meisel, L. Fotiadis, S. Wilker, A. Treytl, and T. Sauter. "Blockchain applications in microgrids an overview of current projects and concepts". In: *43rd Annual Conference of the IEEE Industrial Electronics Society (IECON)*. 2017, pp. 6153–6158. DOI: 10.1109/IECON.2017.8217069.

[160] B. Gramlich. "Smart Contract Languages: A Thorough Comparison". In: *ResearchGate Preprint* (2020). DOI: 10.13140/RG.2.2.22479.92326.

[161] L. Grassi, D. Kales, D. Khovratovich, A. Roy, C. Rechberger, and M. Schofnegger. *Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems*. 2019. Cryptology ePrint Archive: 2019.458.

[162] P. Grau. *Chess on Ethereum*. URL: https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6 (visited on 15. 06. 2020).

[163] M. Green and I. Miers. "Bolt: Anonymous payment channels for decentralized currencies". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 473–489.

[164] J. Groth. "On the size of pairing-based non-interactive arguments". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 305–326.

[165] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. "Updatable and universal common reference strings with applications to zk-SNARKs". In: *Annual International Cryptology Conference*. Springer, 2018, pp. 698–728.

[166] J. Groth and M. Maller. "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks". In: *Annual International Cryptology Conference*. Springer, 2017, pp. 581–612.

[167] J. Groth and A. Sahai. "Efficient non-interactive proof systems for bilinear groups". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2008, pp. 415–432.

[168] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. "Sok: Layer-two blockchain protocols". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 201–226.

[169] I. Gudymenko, A. Khalid, H. Siddiqui, M. Idrees, S. Clauß, A. Luckow, M. Bolsinger, and D. Miehle. "Privacy-preserving Blockchain-based Systems for Car Sharing Leveraging Zero-Knowledge Protocols". In: *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2020, pp. 114–119.

[170] S. Härtlein. "A Blockchain-based Trustless Setup Phase for zkSNARKs". MA thesis. Technische Universität Berlin, 2018.

[171] N. U. Hassan, C. Yuen, and D. Niyato. *Blockchain Technologies for Smart Energy Systems: Fundamentals, Challenges and Solutions*. 2019. arXiv: 1909.02914 [cs.CY].

[172] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. "Sok: General purpose compilers for secure multi-party computation". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1220–1237.

[173] J. Heiss, J. Eberhardt, and S. Tai. "From Oracles to Trustworthy Data On-chaining Systems". In: *2019 IEEE International Conference on Blockchain*. IEEE, 2019.

[174] J. Heiss, M.-R. Ulbricht, and J. Eberhardt. “Put Your Money Where Your Mouth Is – Towards Blockchain-based Consent Violation Detection”. In: *Proceedings of the IEEE International Conference on Blockchain and Cryptocurrency 2020*. IEEE, 2020.

[175] L. Herskind, A. Giaretta, M. De Donno, and N. Dragoni. “BitFlow: Enabling real-time cash-flow evaluations through blockchain”. In: *Concurrency and Computation: Practice and Experience* 32.12 (2020).

[176] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. “Using innovative instructions to create trustworthy software solutions.” In: *HASP@ISCA* 11.10 (2013).

[177] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. *Zcash protocol specification. Version 2020.1.1 [Overwinter+Sapling]*. 2020. URL: https://zips.z.cash/protocol/protocol.pdf (visited on 12. 01. 2020).

[178] J. Horta, D. Kofman, D. Menga, and A. Silva. “Novel market approach for locally balancing renewable energy production and flexible demand”. In: *IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 2017, pp. 533–539. DOI: 10.1109/SmartGridComm.2017.8340728.

[179] L. Hurwicz and S. Reiter. *Designing economic mechanisms*. Cambridge University Press, 2006.

[180] iden3. *circom - Circuit Composition library*. URL: https://github.com/iden3/circom (visited on 06. 11. 2019).

[181] IEEE Standards Association. “Open SystemC language reference manual”. In: *IEEE Std. 1666–2005* (2005).

[182] IEEE Standards Association. “VHDL Language reference manual”. In: *IEEE Std 1076–1987* (1988), pp. 1–218.

[183] Intel. *Innovative Technology for CPU Based Attestation and Sealing*. 2013. URL: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing (visited on 01. 07. 2021).

[184] Intel Corporation. *Intel Software Guard Extensions SDK for Linux OS*. 2020. URL: https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Reference_Linux_2.10_Open_Source.pdf (visited on 10. 07. 2020).

[185] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. “Efficient arguments without short PCPs”. In: *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC’07)*. IEEE, 2007, pp. 278–291.

[186] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. 2017.

[187] S. Jain, P. Saxena, F. Stephan, and J. Teutsch. *How to verify computation with a rational network*. 2016. arXiv: 1606.05917.

[188] M. Jawurek, M. Johns, and F. Kerschbaum. “Plug-In Privacy for Smart Metering Billing”. In: *Privacy Enhancing Technologies*. Springer, 2011, pp. 192–210. DOI: 10.1007/978-3-642-22263-4_11.

[189] C. Jensen and C. Potts. “Privacy Policies as Decision-Making Tools: An Evaluation of Online Privacy Notices”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2004, pp. 471–478. DOI: 10.1145/985692.985752.

[190] Jetbrains. *Meta Programming System (MPS*. URL: https://www.jetbrains.com/mps/ (visited on 06. 11. 2019).
[191] D. Johnson, A. Menezes, and S. Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (2001), pp. 36–63.
[192] S. Jukna. *Boolean function complexity: advances and frontiers*. Vol. 27. Springer Science & Business Media, 2012.
[193] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. "Arbitrum: Scalable, private smart contracts". In: *27th USENIX Security Symposium (USENIX Security)*. USENIX, 2018, pp. 1353–1370.
[194] G. Kalogridis, C. Efthymiou, S. Z. Denic, T. A. Lewis, and R. Cepeda. "Privacy for Smart Meters: Towards Undetectable Appliance Load Signatures". In: *2010 First IEEE International Conference on Smart Grid Communications*. IEEE, 2010, pp. 232–237. DOI: 10.1109/SMARTGRID.2010.5622047.
[195] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
[196] B. W. Kernighan and D. M. Ritchie. *The C programming language*. 2nd ed. Prentice Hall, 1988.
[197] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. 2018. Cryptology ePrint Archive: 2018.642.
[198] A. Kiayias, A. Miller, and D. Zindros. "Non-interactive proofs of proof-of-work". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 505–522.
[199] A. Kiayias and D. Zindros. "Proof-of-work sidechains". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 21–34.
[200] J. Kilian. "A note on efficient zero-knowledge proofs and arguments". In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM, 1992, pp. 723–732.
[201] D.-H. Kim. "Nightlines: Privacy-preserving IOU Tokens". MA thesis. Technische Universität Berlin, 2020.
[202] T. Kim and R. Barbulescu. *Extended Tower Number Field Sieve: A New Complexity for the Medium Prime Case*. 2015. Cryptology ePrint Archive: 2015.1027.
[203] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.
[204] M. Klems, J. Eberhardt, S. Tai, S. Härtlein, S. Buchholz, and A. Tidjani. "Trustless intermediation in blockchain-based decentralized service marketplaces". In: *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 731–739.
[205] D. E. Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.
[206] N. Koblitz and A. J. Menezes. "The random oracle model: a twenty-year retrospective". In: *Designs, Codes and Cryptography* 77.2-3 (2015), pp. 587–610.
[207] C. Konda, M. Connor, D. Westland, Q. Drouot, and P. Brody. *Nightfall*. 2019. URL: https://github.com/EYBlockchain/nightfall/blob/master/doc/whitepaper/nightfall-v1.pdf (visited on 20. 11. 2020).

[208] A. Kosba, C. Papamanthou, and E. Shi. “xJsnark: A Framework for Efficient Verifiable Computation”. In: *IEEE Symposium on Security and Privacy*. IEEE, 2018.

[209] A. Kosba. *jsnark - Java library for Arithmetic Circuit Specification*. URL: https://github.com/akosba/jsnark (visited on 06. 11. 2019).

[210] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 839–858.

[211] B. Kreuter, A. Shelat, B. Mood, and K. Butler. “{PCF}: A Portable Circuit Format for Scalable Two-Party Secure Computation”. In: *22nd USENIX Security Symposium (USENIX Security)*. USENIX, 2013, pp. 321–336.

[212] D. Kuhnert. “Towards Off-chain Computation with Proven Correctness”. MA thesis. Technische Universität Berlin, 2017.

[213] R. Kumaresan, T. Moran, and I. Bentov. “How to use bitcoin to play decentralized poker”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 195–206.

[214] Y. Kwon, J. Liu, M. Kim, D. Song, and Y. Kim. “Impossibility of full decentralization in permissionless blockchains”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM. 2019, pp. 110–123.

[215] L. Lamport et al. “Paxos made simple”. In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[216] A. Langley, M. Hamburg, and S. Turner. “Elliptic curves for security”. In: *Internet Research Task Force RFC* 7748 (2016).

[217] A. Laszka, A. Dubey, M. Walker, and D. Schmidt. “Providing Privacy, Safety, and Security in IoT-based Transactive Energy Systems Using Distributed Ledgers”. In: *Proceedings of the Seventh International Conference on the Internet of Things*. ACM, 2017. DOI: 10.1145/3131542.3131562.

[218] D. V. Le and A. Gervais. *AMR:Autonomous Coin Mixer with Privacy Preserving Reward Distribution*. 2020. arXiv: 2010.01056 [cs.CR].

[219] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, 2020. DOI: 10.1145/3342195.3387532.

[220] D. Leijen. “Division and Modulus for Computer Scientists”. In: (2003). URL: https://www.microsoft.com/en-us/research/publication/division-and-modulus-for-computer-scientists/ (visited on 07. 01. 2021).

[221] C. Li, B. Palanisamy, and R. Xu. *Scalable and Privacy-preserving Design of On/Off-chain Smart Contracts*. 2019. arXiv: 1902.06359.

[222] Z. Li, J. Kang, R. Yu, D. Ye, Q. Deng, and Y. Zhang. “Consortium Blockchain for Secure Energy Trading in Industrial Internet of Things”. In: *IEEE Transactions on Industrial Informatics* (2017). DOI: 10.1109/TII.2017.2786307.

[223] R. Lidl and H. Niederreiter. “Algebraic Foundations”. In: *Introduction to Finite Fields and their Applications*. 2nd ed. Cambridge University Press, 1994, pp. 1–43. DOI: 10.1017/CBO9781139172769.004.

[224] H. Lipmaa. “Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2013, pp. 41–60.

[225] M. Liskov. “Updatable zero-knowledge databases”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2005, pp. 174–198.

[226] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. “Oblivm: A programming framework for secure computation”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 359–376.

[227] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller. “HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and Its Application to Anonymous Communication”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 887–903. DOI: 10.1145/3319535.3354238.

[228] Q. Lu, X. Xu, Y. Liu, I. Weber, L. Zhu, and W. Zhang. “uBaaS: A unified blockchain as a service platform”. In: *Future Generation Computer Systems* 101 (2019), pp. 564–575. DOI: 10.1016/j.future.2019.05.051.

[229] C. Lundkvist, R. Heck, J. Torstensson, Z. Mitton, and M. Sena. “Uport: A platform for self-sovereign identity”. In: (2017). URL: http://blockchainlab.com/pdf/uPort_whitepaper_DRAFT20161020.pdf (visited on 07. 01. 2021).

[230] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, et al. “Fairplay-Secure Two-Party Computation System.” In: *USENIX Security Symposium*. USENIX, 2004.

[231] M. Maller. *A Selection of Pairing-based zkSNARKs*. Zcon1 conference presentation. Split, Croatia, 2019.

[232] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. “Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2111–2128.

[233] K. Mannaro, A. Pinna, and M. Marchesi. “Crypto-trading: Blockchain-oriented energy market”. In: *AEIT International Annual Conference*. IEEE, 2017, pp. 1–5. DOI: 10.23919/AEIT.2017.8240547.

[234] M. M. Mano and M. Ciletti. *Digital design: with an introduction to the Verilog HDL*. Pearson, 2013.

[235] L. Mauri, E. Damiani, and S. Cimato. “Be Your Neighbor’s Miner: Building Trust in Ledger Content via Reciprocally Useful Work”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 53–62. DOI: 10.1109/CLOUD49709.2020.00021.

[236] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller. “Pisa: Arbitration outsourcing for state channels”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM, 2019, pp. 16–30.

[237] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller. “You Sank My Battleship! A Case Study to Evaluate State Channels as a Scaling Solution for Cryptocurrencies”. In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING*

and WTSC, Revised Selected Papers. Springer, 2020, pp. 35–49. DOI: 10.1007/978-3-030-43725-1_4.

[238] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution." In: *Hasp@ISCA* 10.1 (2013).

[239] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. "A fistful of bitcoins: characterizing payments among men with no names". In: *Proceedings of the 2013 Conference on Internet Measurement*. ACM, 2013, pp. 127–140.

[240] E. Mengelkamp, B. Notheisen, C. Beer, D. Dauer, and C. Weinhardt. "A blockchain-based smart grid: towards sustainable local energy markets". In: *Computer Science - Research and Development* 33.1-2 (2018), pp. 207–214. DOI: 10.1007/s00450-017-0360-9. (Visited on 13. 11. 2019).

[241] R. C. Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1987, pp. 369–378.

[242] P. Merriam. *Ethereum Alarm Clock*. URL: https://www.ethereum-alarm-clock.com/ (visited on 11. 01. 2021).

[243] S. Mhanna, A. C. Chapman, and G. Verbič. "A Fast Distributed Algorithm for Large-Scale Demand Response Aggregation". In: *IEEE Transactions on Smart Grid* 7.4 (2016), pp. 2094–2107. DOI: 10.1109/TSG.2016.2536740.

[244] S. Micali. "Computationally sound proofs". In: *SIAM Journal on Computing* 30.4 (2000), pp. 1253–1298.

[245] I. Miers, C. Garman, M. Green, and A. D. Rubin. "Zerocoin: Anonymous distributed e-cash from bitcoin". In: *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 397–411.

[246] M. Mihaylov, S. Jurado, N. Avellana, K. Van Moffaert, I. M. de Abril, and A. Nowé. "NRGcoin: Virtual currency for trading of renewable energy in smart grids". In: *11th International Conference on the European Energy Market (EEM14)*. 2014, pp. 1–6. DOI: 10.1109/EEM.2014.6861213.

[247] M. Mihaylov, I. Razo-Zapata, R. Rădulescu, S. Jurado, and A. Avellana Narcisand Nowé. "Smart Grid Demonstration Platform for Renewable Energy Exchange". In: *Advances in Practical Applications of Scalable Multi-agent Systems. The PAAMS Collection*. Springer, 2016, pp. 277–280.

[248] A. Moghimi, G. Irazoqui, and T. Eisenbarth. *CacheZoom: How SGX Amplifies The Power of Cache Attacks*. 2017. arXiv: 1703.06986.

[249] C. Molina-Jimenez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. Crowcroft. "Implementation of smart contracts using hybrid architectures with on and off–blockchain components". In: *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE, 2018, pp. 83–90.

[250] C. Molina-Jimenez, E. Solaiman, I. Sfyrakis, I. Ng, and J. Crowcroft. "On and off-blockchain enforcement of smart contracts". In: *European Conference on Parallel Processing*. Springer, 2018, pp. 342–354.

[251] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin. “Private memoirs of a smart meter”. In: *Proceedings of the 2nd ACM workshop on Embedded Sensing Systems for Energy-efficiency in Building*. ACM, 2010, pp. 61–66.

[252] Monero. URL: https://web.getmonero.org/ (visited on 01. 11. 2019).

[253] S. Muchnick et al. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[254] R. Mühlberger, S. Bachhofner, E. C. Ferrer, C. Di Ciccio, I. Weber, M. Wöhrer, and U. Zdun. “Foundational Oracle Patterns: Connecting Blockchain to the Off-chain World”. In: *International Conference on Business Process Management*. Springer, 2020, pp. 35–51.

[255] E. Munsing, J. Mather, and S. Moura. “Blockchains for decentralized optimization of energy resources in microgrid networks”. In: *2017 IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, 2017, pp. 2164–2171. DOI: 10.1109/CCTA.2017.8062773.

[256] C. Mussenbrock and S. Karpischek. *Etherisc Whitepaper*. 2018. URL: https://www.etherisc.com/whitepaper (visited on 07. 01. 2021).

[257] M. Naehrig, K. Lauter, and V. Vaikuntanathan. “Can homomorphic encryption be practical?” In: *Proceedings of the 3rd ACM Workshop on Cloud Computing Security*. 2011, pp. 113–124.

[258] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *The Cryptography Mailing List* (2008).

[259] Namecoin. URL: https://www.namecoin.org/ (visited on 01. 11. 2019).

[260] A. Nilsson, P. N. Bideh, and J. Brorsson. *A Survey of Published Attacks on Intel SGX*. 2020. arXiv: 2006.13598 [cs.CR].

[261] O(1) Labs. *Groth16 GPUProver Challenge*. 2019. URL: https://github.com/codaprotocol/gpu-groth16-prover-3x (visited on 02. 05. 2020).

[262] O(1) Labs. *Snarky OCaml DSL*. URL: https://github.com/o1-labs/snarky (visited on 06. 11. 2019).

[263] D. Ongaro and J. Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 2014, pp. 305–319.

[264] OpenZeppelin. *Pull Payment Strategy*. URL: https://docs.openzeppelin.com/contracts/3.x/api/payment#PullPayment (visited on 11. 01. 2021).

[265] F. Pallas. “Beyond gut level — some critical remarks on the German privacy approach to smart metering”. In: *European Data Protection: Coming of Age*. Springer, 2013, pp. 313–345.

[266] F. Pallas. “Data Protection and smart grid communication – The European perspective”. In: *2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*. IEEE, 2012, pp. 1–8.

[267] P. Papadopoulos, I. Q. Azurmendi, J. Zhang, M. Varvello, A. Nappa, and B. Livshits. *ZKSENSE: a Privacy-Preserving Mechanism for Bot Detection in Mobile Devices*. 2019. arXiv: 1911.07649.

[268] B. Parno, J. Howell, C. Gentry, and M. Raykova. “Pinocchio: Nearly practical verifiable computation”. In: *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 238–252.

[269] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. DOI: 10.1002/spe.4380250705.

[270] T. P. Pedersen. “Non-interactive and information-theoretic secure verifiable secret sharing”. In: *Annual International Cryptology Conference*. Springer, 1991, pp. 129–140.

[271] Y. Pei and K. Oida. “An Externally Auditable Identity Management System Using the Bitcoin Blockchain”. In: *Journal of Advances in Information Technology* 9.3 (2018).

[272] Q. Peng and S. H. Low. “Distributed algorithm for optimal power flow on an unbalanced radial network”. In: *2015 54th IEEE Conference on Decision and Control (CDC)*. IEEE, 2015, pp. 6915–6920. DOI: 10.1109/CDC.2015.7403309.

[273] T. Peters. “Algorithms”. In: *Python Cookbook*. O’Reilly Media, Inc., 2002. Chap. 17.

[274] N. Pflugradt. *Load Profile Generator*. URL: https://www.loadprofilegenerator.de/ (visited on 07. 01. 2021).

[275] E. Politou, F. Casino, E. Alepis, and C. Patsakis. “Blockchain mutability: Challenges and proposed solutions”. In: *IEEE Transactions on Emerging Topics in Computing* (2019).

[276] J. Poon and T. Dryja. *The bitcoin lightning network: Scalable off-chain instant payments*. 2015. URL: https://lightning.network (visited on 07. 01. 2021).

[277] C. D. Pop, M. Antal, T. Cioara, I. Anghel, and I. Salomie. “Blockchain and Demand Response: Zero-Knowledge Proofs for Energy Transactions Privacy”. In: *Sensors* 20.19 (2020), pp. 56–78.

[278] Protocol Labs. *Filecoin: A Decentralized Storage Network*. 2017. URL: https://filecoin.io/filecoin.pdf (visited on 15. 02. 2020).

[279] Protocol Labs. *IPFS Documentation — Persistence, Permanence, and Pinning*. URL: https://docs.ipfs.io/concepts/persistence/#pinning-services (visited on 08. 01. 2021).

[280] Provenance Ltd. *Provenance Whitepaper*. 2015. URL: https://www.provenance.org/whitepaper (visited on 07. 01. 2021).

[281] I. A. Qasse, J. Spillner, M. A. Talib, and Q. Nasir. “A Study on ĐApps Characteristics”. In: *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2020, pp. 88–93.

[282] E. L. Quinn. “Privacy and the new energy infrastructure”. In: *SSRN* (2009). DOI: 10.2139/ssrn.1370731.

[283] A. Rastogi, M. A. Hammer, and M. Hicks. “Wysteria: A programming language for generic, mixed-mode multiparty computations”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 655–670.

[284] S. Raval. *Decentralized applications: harnessing Bitcoin’s blockchain technology*. O’Reilly Media, Inc., 2016.

[285] G. D. P. Regulation. “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46”. In: *Official Journal of the European Union (OJ)* 59.1-88 (2016), p. 294.

[286] F. Reid and M. Harrigan. “An analysis of anonymity in the bitcoin system”. In: *Security and Privacy in Social Networks*. Springer, 2013, pp. 197–223.

[287] C. Reitwießner. *From Smart Contracts to Courts with not so Smart Judges - Ethereum Blog*. 2016. URL: https://blog.ethereum.org/2016/02/17/smart-contracts-courts-not-smart-judges/ (visited on 07. 01. 2021).

[288] V. Reniers, D. Van Landuyt, P. Viviani, B. Lagaisse, R. Lombardi, and W. Joosen. "Analysis of Architectural Variants for Auditable Blockchain-Based Private Data Sharing". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2019, pp. 346–354. DOI: 10.1145/3297280.3297316.

[289] A. Rial, G. Danezis, and M. Kohlweiss. "Privacy-preserving smart metering revisited". In: *International Journal of Information Security* 17.1 (2018), pp. 1–31.

[290] C. Riegger, T. Vinçon, and I. Petrov. "Efficient Data and Indexing Structure for Blockchains in Enterprise Systems". In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*. ACM, 2018, pp. 173–182. DOI: 10.1145/3282373.3282402.

[291] D. Ron and A. Shamir. "Quantitative analysis of the full bitcoin transaction graph". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2013, pp. 6–24.

[292] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted execution environment: what it is, and what it is not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. IEEE, 2015, pp. 57–64.

[293] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. "Zerocash: Decentralized anonymous payments from bitcoin". In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 459–474.

[294] J. E. Savage. "Models of computation". In: *Addison-Wesley* (1998).

[295] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

[296] C.-P. Schnorr. "Efficient identification and signatures for smart cards". In: *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 239–252.

[297] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. 2017. arXiv: 1702.08719.

[298] SCIPR Lab. *libsnark - zkSNARK library*. URL: https://github.com/scipr-lab/libsnark (visited on 03. 11. 2020).

[299] SCIPR Lab. *ZEXE - zkSNARK library*. URL: https://github.com/scipr-lab/zexe (visited on 03. 11. 2020).

[300] I. A. Seres, D. A. Nagy, C. Buckland, and P. Burcsi. "Mixeth: efficient, trustless coin mixing service for ethereum". In: *International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019)*. 2019.

[301] S. Setty. "Spartan: Efficient and general-purpose zkSNARKs without trusted setup". In: *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.

[302] S. Setty and J. Lee. *Quarks: Quadruple-efficient transparent zkSNARKs*. 2020. Cryptology ePrint Archive: 2020.1275.

[303] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. "Taking proof-based verified computation a few steps closer to practicality". In: *21st USENIX Security Symposium (USENIX Security)*. 2012, pp. 253–268.

[304] A. Shahnaz, U. Qamar, and A. Khalid. “Using Blockchain for Electronic Health Records”. In: *IEEE Access* 7 (2019). DOI: 10.1109/ACCESS.2019.2946373.

[305] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176.

[306] A. Shamir, R. L. Rivest, and L. M. Adleman. “Mental poker”. In: *The mathematical gardner*. Springer, 1981, pp. 37–43.

[307] B. Sharma, R. Halder, and J. Singh. “Blockchain-based Interoperable Healthcare using Zero-Knowledge Proofs and Proxy Re-Encryption”. In: *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 2020, pp. 1–6.

[308] L. Shin. “The first government to secure land titles on the bitcoin blockchain expands project”. In: *Forbes Mag., Feb* 7 (2017).

[309] I. Shparlinski. *Finite Fields: Theory and Computation: The meeting point of number theory, computer science, coding theory and cryptography*. Vol. 477. Springer Science & Business Media, 2013.

[310] A. Shpilka, A. Yehudayoff, et al. “Arithmetic circuits: A survey of recent results and open questions”. In: *Foundations and Trends in Theoretical Computer Science* 5.3–4 (2010), pp. 207–388.

[311] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. “Tinygarble: Highly compressed and scalable sequential garbled circuits”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 411–428.

[312] A. Stanley. “Ready to rumble: IBM launches food trust blockchain for commercial use”. In: *Forbes Mag., Oct* (2018).

[313] State of the dApps - Decentralized Application Registry. URL: https://www.stateofthedapps.com/ (visited on 10. 10. 2020).

[314] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev. “zkay: Specifying and Enforcing Data Privacy in Smart Contracts”. In: *2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019. DOI: 10.1145/3319535.3363222.

[315] O. Stengele and H. Hartenstein. “Atomic Information Disclosure of Off-Chained Computations Using Threshold Encryption”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2018, pp. 85–93.

[316] P. Šulc, S. Backhaus, and M. Chertkov. “Optimal Distributed Control of Reactive Power Via the Alternating Direction Method of Multipliers”. In: *IEEE Transactions on Energy Conversion* 29.4 (2014), pp. 968–977. DOI: 10.1109/TEC.2014.2363196.

[317] N. Szabo. “The idea of smart contracts”. In: *Nick Szabo’s Papers and Concise Tutorials* 6 (1997).

[318] S. Tai. “Continuous, trustless, and fair: Changing priorities in services computing”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer, 2016, pp. 205–210.

[319] S. Tai, J. Eberhardt, and M. Klems. “Not ACID, not BASE, but SALT – A Transaction Processing Perspective on Blockchains”. In: *Proceedings of the International Conference on Cloud Computing and Services Science*. ScitePress, 2017, pp. 755–764. DOI: 10.5220/0006408207550764.

[320] J. Teutsch and C. Reitwießner. *A scalable verification solution for blockchains*. 2017. arXiv: 1908.04756 [cs.CR].

[321] J. Teutsch, M. Straka, and D. Boneh. *Retrofitting a two-way peg between blockchains*. 2019. arXiv: 1908.03999 [cs.CR].

[322] The Initiative for Cryptocurrencies and Contracts (IC3). *HoneyBadgerMPC Ethereum Integration*. 2020. URL: https://github.com/initc3/HoneyBadgerMPC/tree/ethbadgermpc (visited on 01. 12. 2020).

[323] D. Thomas and P. Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[324] V. Trón, A. Fischer, D. A. Nagy, Z. Felföldi, and N. Johnson. *Swap, Swear and Swindle - Incentive System for Swarm*. 2016. URL: https://ethersphere.github.io/swarm-home/ethersphere/orange-papers/1/sw%5E3.pdf (visited on 12. 11. 2020).

[325] *Truffle - Ethereum Development Framework*. URL: https://www.trufflesuite.com/ (visited on 15. 02. 2020).

[326] S. Tsai, Y. Tseng, and T. Chang. "Communication-Efficient Distributed Demand Response: A Randomized ADMM Approach". In: *IEEE Transactions on Smart Grid* 8.3 (2017), pp. 1085–1095. DOI: 10.1109/TSG.2015.2469669.

[327] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936).

[328] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. "SoK: secure messaging". In: *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 232–249.

[329] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium (USENIX Security)*. USENIX, 2018, pp. 991–1008.

[330] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[331] Visa. URL: https://usa.visa.com/run-your-business/small-business-tools/retail.html (visited on 04. 11. 2020).

[332] F. Vogelsteller and V. Buterin. *ERC-20 Token Standard*. 2015. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md (visited on 06. 11. 2020).

[333] M. Vukolić. "Rethinking Permissioned Blockchains". In: *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, 2017, pp. 3–7. DOI: 10.1145/3055518.3055526.

[334] R. S. Wahby, S. T. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. "Efficient RAM and control flow in verifiable outsourced computation." In: *Network & Distributed System Security Symposium (NDSS)*. Internet Society, 2015.

[335] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. "Doubly-efficient zkSNARKs without trusted setup". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 926–943.

[336] M. Walfish and A. J. Blumberg. “Verifying Computations Without Reexecuting Them”. In: *Commun. ACM* 58.2 (2015), pp. 74–84. DOI: 10.1145/2641562.

[337] G. Wang, Z. J. Shi, M. Nixon, and S. Han. “SoK: Sharding on Blockchain”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. ACM, 2019, pp. 41–61.

[338] M. Wang and Q. Wu. *Lever: Breaking the Shackles of Scalable On-chain Validation*. 2019. Cryptology ePrint Archive: 2019.1172.

[339] W. Wang and Z. Lu. “Cyber security in the Smart Grid: Survey and challenges”. In: *Computer Networks* 57.5 (2013), pp. 1344–1371. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2012.12.017.

[340] Y. Wang, L. Wu, and S. Wang. “A Fully-Decentralized Consensus-Based ADMM Approach for DC-OPF With Demand Response”. In: *IEEE Transactions on Smart Grid* 8.6 (2017), pp. 2637–2647. DOI: 10.1109/TSG.2016.2532467.

[341] WebAssembly Community Group. *WebAssembly Core Specification*. 2019. URL: https://www.w3.org/TR/wasm-core-1/ (visited on 07. 01. 2021).

[342] M. Westerkamp. “Verifiable Smart Contract Portability”. In: *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2019, pp. 1–9. DOI: 10.1109/BLOC.2019.8751335.

[343] M. Westerkamp and J. Eberhardt. “zkRelay: Facilitating Sidechains using zkSNARK-based Chain-Relays”. In: *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*. IEEE, 2020, pp. 378–386. DOI: 10.1109/EuroSPW51379.2020.00058.

[344] D. Westphal. “Implementing Private Transaction on the Ethereum Blockchain”. MA thesis. Technische Universität Berlin, 2019.

[345] B. Whitehat. *Baby jubjub prototype*. URL: https://github.com/barryWhiteHat/baby_jubjub_ecc (visited on 09. 09. 2019).

[346] B. WhiteHat, J. Baylina, and M. Bellés. *Baby Jubjub Elliptic Curve*. URL: https://iden3-docs.readthedocs.io/en/latest/_downloads/33717d75ab84e11313cc0d8a090b636f/Baby-Jubjub.pdf (visited on 09. 09. 2019).

[347] R. Wilhelm, H. Seidl, and S. Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.

[348] M. Wöhrer and U. Zdun. “Design patterns for smart contracts in the ethereum ecosystem”. In: *IEEE International Conference on Blockchain*. IEEE, 2018, pp. 1513–1520.

[349] M. Wöhrer and U. Zdun. “Smart contracts: security patterns in the ethereum ecosystem and solidity”. In: *International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

[350] G. Wood. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum Project Yellow Paper* (2014).

[351] G. Wood and Y. e. a. Hirai. “Ethereum Yellow Paper, BYZANTIUM VERSION 2018-03-12, Appendix E.” In: *Ethereum Project Yellow Paper* (2018).

[352] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. “{DIZK}: A distributed zero knowledge proof system”. In: *27th USENIX Security Symposium (USENIX Security)*. USENIX. 2018, pp. 675–692.

[353] K. Wu. *An empirical study of blockchain-based decentralized applications*. 2019. arXiv: 1902.04969.

[354] K.-W. Wu, S. Y. Huang, D. C. Yen, and I. Popova. "The effect of online privacy policy on consumer privacy concern and trust". In: *Computers in Human Behavior* 28.3 (2012), pp. 889–897.

[355] K. Wüst and A. Gervais. "Do you need a Blockchain?" In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 2018, pp. 45–54.

[356] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *Annual International Cryptology Conference*. Springer, 2019, pp. 733–764.

[357] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber. "A pattern collection for blockchain-based applications". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. ACM, 2018, p. 3.

[358] A. C. Yao. "Protocols for secure computations". In: *23rd Annual Symposium on Foundations of Computer Science*. IEEE, 1982, pp. 160–164.

[359] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt. *SoK: Communication Across Distributed Ledgers*. 2019. Cryptology ePrint Archive: 2019/1128.

[360] A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, and W. J. Knottenbelt. "A wild velvet fork appears! inclusive blockchain protocol changes in practice". In: *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 31–42.

[361] *Zcash*. URL: https://z.cash/ (visited on 06. 12. 2020).

[362] Zcash. *Zcash Sapling Circuit implementation*. URL: https://github.com/zcash-hackworks/sapling-crypto (visited on 09. 05. 2020).

[363] Zcash. *Zcash Sprout MPC Rust implementation*. URL: https://github.com/zcash/mpc (visited on 20. 04. 2020).

[364] *Zcash Sapling Upgrade*. URL: https://z.cash/upgrade/sapling/ (visited on 12. 03. 2020).

[365] P. Zhang, J. White, D. C. Schmidt, and G. Lenz. "Design of blockchain-based apps using familiar software patterns to address interoperability challenges in healthcare". In: *24th Conference On Pattern Languages Of Programs*. ACM, 2017.

[366] Y. Zhu, X. Song, S. Yang, Y. Qin, and Q. Zhou. "Secure Smart Contract System Built on SMPC Over Blockchain". In: *IEEE International Conference on Blockchain*. IEEE, 2018, pp. 1539–1544.

[367] G. Zyskind, O. Nathan, and A. Pentland. *Enigma: Decentralized Computation Platform with Guaranteed Privacy*. 2015. arXiv: 1506.03471.

List of Abbreviations

ABI	application binary interface	p. 156
AIR	algebraic intermediate representation	p. 66
ALU	arithmetic logic unit	p. 116
API	application programming interface	p. 157
ASIC	application-specific integrated circuit	p. 228
AST	abstract syntax tree	p. 145
CAS	content-addressable storage	p. 43
CI/CD	continuous integration and delivery	p. 171
CLI	command-line interface	p. 153
CPU	central processing unit	p. 190
CRS	common reference string	p. 23
DAG	directed acyclic graph	p. 116
dApp	decentralized application	p. 4
DSL	domain-specific language	p. 32
ECDH	elliptic curve Diffie-Hellman key exchange	p. 140
ECDSA	elliptic curve digital signature algorithm	p. 140
EVM	Ethereum Virtual Machine	p. 18
FHE	fully homomorphic encryption	p. 20
GPU	graphics processing unit	p. 190
GUI	graphical user interface	p. 171
HPU	household processing unit	p. 203
IDE	integrated development environment	p. 170
ISA	instruction set architecture	p. 116
JSON	JavaScript Object Notation	p. 157
LOC	lines of code	p. 236
MPC	secure multi-party computation	p. 23
NFT	non-fungible token	p. 231
NIPoPoW	Non-Interactive Proofs of Proof-of-Work	p. 228
PCP	probabilistically checkable proof	p. 20
PEG	Parsing Expression Grammar	p. 119
PoS	Proof-of-Stake	p. 228

PoW	Proof-of-Work	p. 15
QAP	quadratic arithmetic program	p. 25
QoS	Quality of Service	p. 186
R1CS	rank-1 constraint system	p. 22
RAM	random access machine	p. 21
SLP	straight-line program	p. 95
SPV	simplified payment verification	p. 6
SSA	static single-assignment form	p. 149
TEE	trusted execution environment	p. 30
Wasm	WebAssembly	p. 75
ZIR	ZoKrates intermediate representation	p. 10
zk-SNARK	zero-knowledge succinct non-interactive argument of knowledge	p. 10
zk-STARK	zero-knowledge scalable transparent arguments of knowledge	p. 24
ZTF	ZoKrates text format	p. 84

List of Figures

1.1 Thesis Organization 12
1.2 How to Read & Dependencies 13

4.1 Content-Addressable Storage Pattern Example 43
4.2 Comparison of On-chain Processing and Verifiable Off-chain Computations . . 46
4.3 Off- and On-chain Interactions in the Chess Application Example 49
4.4 Flowchart representing the Optimistic Finalization Process 51
4.5 Optimistic Finalization Pattern applied to Chess Example 52

5.1 Basic Off-chain Computation Architecture 61
5.2 Off-chain Computation Protocol Dimensions and Manifestations 62
5.3 Architecture of Systems that realize the Verifiable Off-chain Computation Approach 63
5.4 Schematic Overview of the Enclave-based Off-chain Computation Approach . . 67
5.5 Schematic Overview of the Incentive-driven Off-chain Computation Approach . 68
5.6 Schematic Overview of the MPC-based Off-chain Computation Approach . . . 71

6.1 Gap between Problem Domains. 82
6.2 ZoKrates Process 83
6.3 ZoKrates Development in the Remix IDE 89

7.1 Prime Field Example with $p = 13$ 94
7.2 Example Arithmetic Circuit 95

8.1 Frontend-part of ZoKrates Compilation Process: `.zok` to ZIR. 144
8.2 ZoKrates Constraint Generation Process: ZIR to Backend-specific Representation 145

9.1 Overview of the ZoKrates Architecture 154

10.1 Overview of the ZoKrates Codebase 166
10.2 Overview of Parsing Implementation 168
10.3 ZoKrates Remix Plugin User Interface Example 171

11.1 On-chain Verification Cost in Gas per # of Inputs 186
11.2 Verification Contract Deployment Cost in Gas per # of Inputs 188

12.1 Local Distribution Grid formed by Households and Supplier. 200
12.2 Actors and their Relation in the Energy Market 200
12.3 Conceptual Architecture without Privacy Protection. 203
12.4 Conceptual Architecture with Privacy Protection. 205
12.5 Proof-of-Concept Implementation Architecture. 206
12.6 User Interface presented to a Household. 208
12.7 Overview of the Netting Process. 210
12.8 Example Sequence of the Netting Algorithm 214
12.9 Program Execution and Proving Time for the ZoKrates Program. 215
12.10On-chain Verification Cost of a Netting Result. 216

13.1 Overview of the zkRelay System 220
13.2 zkRelay Epoch Batch Validation ($N = L = 2016$) 222
13.3 zkRelay Verification of Batches with Flexible Size 223
13.4 Runtime and Memory Consumption of Off-chain Validation and Proof Generation227

14.1 Overview of Nightfall's Core Components 233

List of Tables

2.1 Categorization of Blockchain Deployments by Permission including Examples 19

5.1 Comparison of Off-chain Computation Approaches 73

7.1 Example Prime Field Arithmetics for $p = 13$ 93
7.2 Truth Tables for Functionally Complete Set of Boolean Operators 104

8.1 Arithmetic Operators 128
8.2 Comparison Operators 129
8.3 Equivalence Proof for Relational Operator Transformations 131
8.4 Logic Operators 131
8.5 Logic Operations as Arithmetic Expressions 131
8.6 Equivalence Proof for Logic Operator Transformations 132
8.7 Bitwise Operators 132
8.8 Miscellaneous Operators 132
8.9 Precendence Rules and Associativity of ZoKrates Operators 133

9.1 Available Verifiable Computation Schemes and Supporting Backends 160

11.1 Categorization of Evaluated ZoKrates Operations 174
11.2 Time and Memory Consumption for Witness Computation and Proof Generation Steps 177
11.3 Time and Memory Consumption for Compilation and Setup Steps 179
11.4 Impact of Compiler Optimizations on Constraint Count and Processing Times . 182
11.5 On-chain Verification Cost Estimation for Ethereum Mainnet 187
11.6 Verification Smart Contract Deployment Cost Estimation for Ethereum Mainnet 189

13.1 Comparison of different Sidechain Mechanisms 228

14.1 Overview of ZoKrates Programs used in Nightfall Implementation 236
14.2 ZoKrates Program Metrics & Performance of Off-chain Steps 236
14.3 ZoKrates-related Message Sizes and On-chain Processing Cost 237

List of Listings

6.1 ZoKrates Program Computing `SHA-256`-Hash 84
6.2 Solidity Verification Smart Contract for `SHA-256` Program 86
6.3 Proof and Inputs in JSON Format 88

7.1 Example Arithmetic Circuit in ZoKrates 113
7.2 ZoKrates Internal Representation compiled from Example Arithmetic Circuit . 113

8.1 A First ZoKrates Program 120
8.2 ZoKrates Scoping Examples 122
8.3 Primitive ZoKrates Type Declaration and Initialization 122
8.4 Usage Examples for the ZoKrates Array Type 125
8.5 Example ZoKrates Program Using the Struct Data Type 127
8.6 ZoKrates Functions Example 135
8.7 ZoKrates For-Loop Example 136
8.8 ZoKrates Conditionals Example 137
8.9 ZoKrates Modules and Imports Example 138
8.10 ZoKrates Module `bar.zok` 138
8.11 ZoKrates Module `foo.zok` 139
8.12 ZoKrates Module `point.zok` 139
8.13 BabyJubJub Point Addition in ZoKrates 141

9.1 Point Coordinate Inversion ZoKrates Program 158
9.2 Point Coordinate Inversion ZIR in ZTF 158
9.3 ZIR ABI Description for Point Inversion ZoKrates Program 159
9.4 JSON Inputs for Point Inversion ZIR Program 159
9.5 JSON Outputs of Point Inversion ZIR Program 160
9.6 Solidity Contract Excerpt of Verification Logic for Groth16 [164] 163

A.1 ZoKrates Parsing Expression Grammar 245