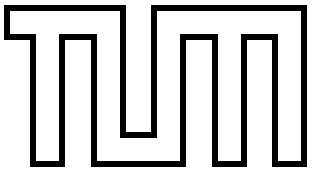


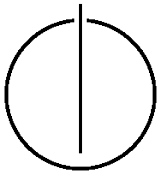
DEPARTMENT OF INFORMATICS

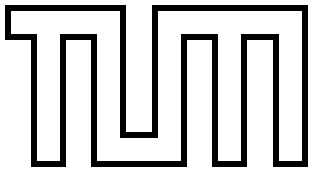
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis and Implementation of Verifiable Computation Techniques for Energy Blockchain Applications

Bernd Steinkopf





DEPARTMENT OF INFORMATICS

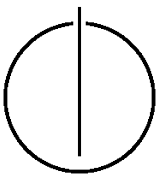
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Analysis and Implementation of Verifiable Computation Techniques for Energy Blockchain Applications

Analyse und Implementierung verifizierbarer Berechnungstechniken für energiewirtschaftliche Blockchain-Anwendungen

Author:	Bernd Steinkopf
Supervisor:	Prof. Dr. Florian Matthes
Advisors:	Ulrich Gallersdörfer, Andreas Zeiselmair
Date:	December 15th, 2020



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, December 15th, 2020 Bernd Steinkopf

Acknowledgments

Most importantly, I would like to thank Professor Dr. Florian Matthes and the entire SEBIS chair for providing me with the opportunity to write my thesis in such an exciting and future-oriented field of research.

Next, I would also like to thank Ulrich Gallersdörfer for his continued feedback and helping me navigate the waters of institutional bureaucracy.

Finally, I would like to thank my advisor Andreas Zeiselmair for his guidance and support throughout these six months. His insights and deep understanding of the field have been a vital contribution to this work.

Abstract

With the invention of smart contracts, many industries look to incorporate the generous security guarantees of blockchains into their business processes. Especially the energy sector would benefit greatly from the increased reliability and transparency of a fully decentralized network. Unfortunately, many of its use cases rely on computationally expensive tasks that are fundamentally unsuitable for such a slow and highly redundant environment. In this thesis, we explore different strategies for making these use cases possible without compromising the advantages offered by using a blockchain.

In the first part, we examine three different technologies for outsourcing expensive work from computationally weak nodes in a secure manner. Collectively known under the name *verifiable computing*, they are (in order of appearance): *trusted oracles*, *zero-knowledge proofs* and *secure multi-party computation*. We provide theoretical background knowledge for all three approaches and discuss some of the most popular state-of-the-art implementations. Each individual technology is further analyzed for its strengths and weaknesses regarding multiple categories such as security, performance or usability.

In the second, practical part of this work, we implement a zkSNARK, one of the identified technologies that was found to be most promising during the aforementioned evaluation. We give an overview of the employed algorithms and technology-specific challenges encountered along the way. To summarize, our implementation is capable of solving many of the optimization problems that commonly occur in the energy industry.

The finished program is subsequently tested for performance under different conditions, showing good results for small-scale optimizations but quickly deteriorating in performance for bigger problems. We conclude from our findings, that verifiable computing is an important and necessary field but still requires a concerted effort from researchers and engineers alike to become a truly viable alternative to traditional untrusted computations.

Contents

Outline of the Thesis

Part I: Introduction

1. Motivation

This chapter presents the motivation for this work. It reflects on the current state of blockchain applications in the energy industry and why there is still ample room for improvement.

2. Outline

This chapter outlines the structure of this work in more detail. It also discusses related literature.

Part II-IV: Verifiable Computing Techniques

1. Concept

Here, we give a general overview of the technology and its core principles.

2. Background

In this chapter, we take an in-depth look at the theoretical frameworks, which support the technology. Mathematical models are explained and necessary abstractions formulated.

3. Implementations

This chapter deals with real-world implementations of the specified technology. We list current market leaders and explore their architectural differences.

Part V: Evaluation

1. Method

Initially, we detail the evaluation framework which we use to examine the identified technologies.

2. Inter-Technology Comparison

Then, we compare different realizations within one technology group, applying the framework which we developed in the previous chapter.

3. Use Case Suitability

Finally, we choose the technology which is most suitable for our implemented use case, based on our assessment.

Part VI: Implementation

Part I.

Introduction

1. Motivation

With the introduction of smart contracts, the blockchain ecosystem has experienced a dramatic paradigm shift. The technology has been elevated from being a mere digital currency to providing secure and trusted general-purpose computations which can be verified by anyone [24]. Since then, smart contract platforms have found widespread adoption in many industry sectors, e.g. supply-chain management, finance, healthcare, and others [26]. Among the most critical applications of smart contracts, however, is a transparent and secure supply of power. Indeed, the energy sector is one of the most rapid adopters of blockchain technology [97]. Its distributed nature, high security standards and calls for more transparency make it an ideal fit.

Unfortunately, smart contracts are a relatively recent invention and as such are plagued by a host of different problems that make their use in production systems less than ideal. All blockchain platforms suffer from poor transaction throughput. An often quoted statistic in this context is comparing VISA's roughly 2,000 transactions per second to Ethereum's 15 [53]. They also achieve their strong security claims by compromising user privacy. All transactions are sent in the clear and an attacker could easily trace them back to their originators [32]. While these factors affect all industries in some way, the energy sector's large scale and critical role in a society's infrastructure make it especially susceptible. Many use cases in the energy sector rely on high-performance calculations on sensitive data (conf. 15.1) and integrating them into a blockchain environment at this point in time would prove difficult.

The blockchain community is acutely aware of the shortcomings of current smart contract implementations and has been working on several solutions. Most of their effort has been directed towards improving the transaction throughput of major smart contract chains. Two of the most promising technologies have even been suggested by the developers of the biggest smart contract chain itself – Ethereum.

- *Sharding* [36] refers to a technique where the original blockchain is split into multiple "shards". Within each shard, state updates are propagated as usual, but communication between shards is limited to a simple synchronization mechanism. This way, shard data can be processed in parallel, significantly increasing the number of transactions per second.

- *Plasma* [84] is an integrated network of micro-channels for Ethereum. A micro-channel condenses multiple transactions into one and only writes the final outcome back to chain. Its validity is then checked via a "fraud proof", an interactive verification algorithm where anyone can participate. With this technique, the required number of total transactions can be dramatically reduced.

Considerably fewer solutions address the problem of privacy-preserving smart contracts. Classical techniques from cryptography, e.g. asymmetric encryption or hash- and

reveal-schemes, can be used to secure private contract data, but their applications are limited and often require extensive key infrastructures already in place. Therefore, many critical industries opt instead to use private blockchains with restricted access to store their sensitive data [68]. While this is a valid approach, it diminishes many of the initial advantages of the original blockchain idea. Open participation and distributed consensus are the very elements which give blockchains their strong security model.

As of recently, a comprehensive third approach has emerged, which tries to combine both performance and privacy. The ideas of *verifiable computing* are relatively old but their particular application to blockchain environments has once again brought them to the attention of researchers worldwide. The general concept is similar to Plasma's microchannels but instead of merely aggregating transactions, the entire smart contract is executed offline and its results are published. A verification algorithm subsequently ensures that the result was computed correctly. While not strictly defined that way, most verifiable computing schemes also keep the contract's data private during all of this. Thus, verifiable computing is the perfect tool to finally bring the security of blockchain to the various sensitive and demanding uses cases of the energy sector.

2. Outline

The goal of this work is to implement and evaluate a blockchain-compatible solution for verifiable computation that can meet the high standards imposed by energy-industrial applications. For this, we have formulated the following research questions:

1. "What are the requirements and challenges of common blockchain applications in the energy industry?"
2. "How can the use of verifiable computing increase the security and reliability of energy-related blockchain applications?"
3. "How do different blockchain-based verifiable computing techniques compare in terms of performance, security and usability?"
4. "Which measures are required to prototypically implement a functioning infrastructure for solving blockchain-aided verifiable computations?"

In the following, we will discuss our approach in answering these questions and outline the topics which lie ahead.

Contents This work consists of two distinct parts. The former, theoretical part lays the foundation for our understanding of verifiable computing. In it, we provide an in-depth discussion of the three main approaches to verifiable computing which are common today. In addition to explaining the inner workings of each technology on an abstract level, we give theoretical background information and take a look at the most popular implementations which are currently available on the market. The theory sections then concludes with an evaluation of the three technologies and directly compares performance, security and usability of each one [V]. In the latter part, we apply our newfound knowledge and implement a blockchain use case relevant to the energy industry using one of the discussed approaches [VI]. After running and evaluating our implementation, we finish by discussing the obtained results and draw possible conclusions [17].

Our initial overview starts with *trusted oracles* [II], a concept that is likely familiar to those with a blockchain background. Trusted oracles leverage the capabilities of existing blockchain frameworks and apply them to verifiable computing through various means, such as game theory or cryptography. In III, we continue delving further into recent advancements in cryptography by taking a closer look at zero-knowledge proofs and zk-SNARKs specifically. In short, zero-knowledge proofs promise being able to generate a mathematical proof for any arbitrary computation and later allow anyone possessing this proof to verify that computation's correctness. Seeing how this would solve verifiable computing once and for all, it is hardly surprising that this technology is one of the blockchain community's principle research topics. Lastly, we take a look at the near future with MPC [IV]. This technology goes even further than zero-knowledge proofs and

promises that verifiable computing can happen in a distributed manner, much in the spirit of blockchain itself.

Related Work There exists much literature that is specific to each of our three discussed technologies. This includes original specifications, improvement proposals, evaluations and comparisons. We list these works in their respective sections. In regards to analyzing blockchain-compatible verifiable computing on an inter-technological level, we are only aware of one other work by Eberhardt et al. [39]. In it, the authors were able to identify mostly the same categories of verifiable computing but discuss each approach at a much lower level of detail. As an article paper, their work naturally lacks any in-depth explanation of the different theoretical backgrounds and what real-world solutions are currently available. Additionally, the work remains purely theoretical, as the authors do not provide an implementation for any of the listed categories. Hence, we hope to contribute to their research by filling in some the mentioned gaps herein.

Part II.

Trusted Oracles

3. Concept

Oracles are a relatively old and well researched part of the blockchain ecosystem. In this chapter, we explain why they still pose one of the biggest unsolved challenges for smart contract practicability. We then take look at how some of the presented solutions can be applied to verifiable computing.

Blockchain Oracles One of the greatest limitations of smart contracts is their fully deterministic nature. When a contract's code is invoked, a majority of the blockchain's participants must agree on its result. This is a necessary requirement of the underlying consensus algorithm. If smart contracts were allowed to yield a randomized result for every participant, such a majority could not be guaranteed.

Perhaps counterintuitively, this absence of non-determinism is not always ideal. A consequence of particular significance is the fact, that smart contracts cannot communicate with foreign APIs, since both, the network code and the API itself might exhibit non-deterministic behavior. Many potential blockchain use-cases are handicapped by this restriction, since they cannot depend on data from outside sources. Blockchain oracles [2] aim to mitigate this problem by on-chaining the data in a way that can be reconciled by the consensus algorithm.

All oracle implementations must contain at least one dedicated smart contract, so that they can interact with the blockchain. This contract serves as a central hub where outside information is gathered and forwarded to other smart contracts. To inject data into the system, it must first be retrieved from the outside source and subsequently included in a transaction. The transaction is then sent to the oracle contract, which either stores it in a public field or passes it on to any interested parties.

Packaging and transferring the data can either be done manually by a human operator or by an automated script. There are multiple implementation-specific ways to trigger an injection. Common ones include other smart contracts directly requesting the data, time-based injections, event-based injections, etc.

Trusted Oracles The simple oracle architecture we have provided above is already capable of successfully on-chaining outside data. Nonetheless, it suffers from one major flaw. A smart contract's main advantage over traditional programs is the guarantee that its code will be executed exactly as advertised. Users can rely on the blockchain's consensus algorithm to identify and remove any incorrect results.

With outside data, however, the concept of a "correct result" is not clearly defined, since it appears as a black box to on-chain contracts. Hence, the consensus algorithm can only verify that an injection occurred and not the data itself. In literature, this observation is referred to as the *Oracle Problem* [41].

The Oracle Problem makes naive implementations an easy target for manipulation attempts. For example, an oracle might claim to provide the current price of Bitcoin in US-Dollars, but when a smart contract calls the service, it receives a number that is higher than the real value. Depending on the calling smart contract, such an attack may incur severe financial losses for the user.

In practice, several approaches exist that endow oracles with a notion of trust akin to the one presented by blockchains. They range from simple redundancy checks to more sophisticated incentive schemes and even the use of advanced cryptographic hardware (conf. 4).

Because trusted oracles can check the correctness of their data before it is sent to the blockchain, they make an excellent tool for verifiable computations [52]. Users can simply transmit their computations to an oracle service, where they are executed off-chain. After the computation has finished, the service provider on-chains the result via the oracle's contract. The trust mechanism employed by the oracle service lets users know that the result is genuine. As long they accept the correctness of the mechanism, they can also accept the result.

4. Background

In the following chapter, we outline two different technologies which find widespread use when implementing trust in oracles. We chose these particular approaches because they were specifically designed with the verifiable computation paradigm in mind.

4.1. Incentive-driven off-chain computation

Incentive-driven off-chain computation (IOC) [52, 39] is an approach to oracle-based off-chaining which has its origins in game theory. Instead of employing cryptographic techniques, an IOC system relies on reward and punishment rules to establish the correctness of computational results.

In essence, the same computation is replicated over a number of participating oracle nodes. When all nodes have finished their work, they then vote on the correct outcome by submitting their result. The majority vote is eventually accepted as the sole solution.

As their name implies, IOC systems employ an incentive scheme to enforce truthful behavior. Participants who publish the majority result receive rewards, which are usually monetary in nature. Minority voters either receive nothing or might even be punished for their failure to comply with the protocol. This incentive-driven voting game is based on many of the same principles found in the proof-of-work consensus algorithm used by early blockchain implementations [77].

The security of such a system always rests on the assumption that there is an honest majority who will vote truthfully. To ensure the existence of an honest majority, the chosen reward scheme must be *incentive compatible*. This game theoretic term describes any system where participants must act in accordance with their true preference if they wish to maximize their profits.

In an IOC setting most participating oracles have no personal preference for the outcome of a given computation, but they will instead answer truthfully, as this gives them the highest chance of being part of the majority. This is the case since each participant will naturally believe their result to be correct and they have no reason to assume that the other participants will answer incorrectly. As long as the network stays large enough, any party intentionally trying to manipulate the outcome of a computation will therefore be dwarfed by the votes of the honest players.

4.2. Intel SGX

Software Guard Extensions (SGX) is a collection of cryptographic co-processors developed and introduced by Intel [75, 54, 33]. It provides a secure container where application code can run in isolation from the rest of its host system. To achieve this, SGX enforces code

confidentiality and integrity at a hardware level during runtime. Not even privileged software, such as the OS or the BIOS, can access the program's memory contents.

Application developers can use SGX to implement strong security mechanisms without needing access to advanced knowledge in cryptography. In the blockchain space, SGX is already a de-facto standard and can be found in popular applications such as Hyperledger Sawtooth [80] and many trusted oracles (conf. 5).

4.2.1. Theory

Intel SGX is a slightly modified adaptation of the *Trusted Execution Environment (TEE)* concept [9, 87]. The general idea behind TEEs is the ability to run computations remotely, without requiring trust in the remote node's underlying system. TEEs are thus a logical evolution of *Trusted Platform Modules (TPM)*.

Security Guarantees One of the chief security guarantees that TEEs make, is the continued confidentiality and integrity of their associated applications. Therefore, a secure memory abstraction lies at the core of every TEE implementation. There, the program's code is contained during runtime.

Any successful TEE must ensure that this memory region cannot be read from or written to by unauthorized software. Should an outsider gain the ability to read from it, they have unlimited access to potentially sensitive data. Similarly, should they gain write-access, CPU instructions could be altered on the fly, leading to unwanted application behavior. Both cases pose obvious security risks that are unacceptable for critical applications.

TEEs further guarantee the correct behavior of their contained application. To this end, they closely monitor the execution flow of their host system. A host is only allowed to interact with its TEE over well-defined APIs. This greatly limits the TEE's attack surface.

It is also the TEE's job to sanitize the system state whenever execution flow enters or leaves its trusted environment. Modern host systems often have complex hardware architectures with many caches and buffers to increase their computation speed. It is essential that these caches be sanitized by the TEE, so that no confidential information can be leaked by an intruder.

Lastly, most TEEs also provide a protocol allowing third parties to verify their integrity over a remote connection. This process is colloquially known as *attestation*. Attestation combines all of the other security guarantees by bundling their artifacts into a publicly verifiable certificate. Anyone in possession of this certificate has the ability to convince themselves of the TEE's correct operation.

Technical Realization TEEs typically rely on a combination of advanced cryptographic techniques to enforce their security guarantees. Code confidentiality is usually achieved through encryption, integrity via cryptographic hash functions. Attestation, on the other hand, is mostly based on public-key signature schemes. Controlled execution flow can either be implemented in hardware or software.

Being the most privileged agent of any platform, the CPU plays a critical role in most hardware approaches. The go-to solution here is using an elevated CPU mode to clearly

separate TEE execution from work load incurred by regular user activity. For purely software based solutions, this isolation can be simulated by a specialized security kernel that is part of the OS. However, even in the case of these software-based TEEs, additional cryptographic co-processors are often needed to store encryption keys and other confidential information.

Unlike their predecessor, TEEs have not yet been standardized as of the time of writing. First plans for standardization have been put forward by a number of entities. Most notably, GlobalPlatform drives an initiative to standardize TEEs for use in industrial-grade IoT devices [48]. The task, however, remains difficult due to the relative immaturity of the technology and the heterogeneous nature of existing implementations.

4.2.2. Architecture

SGX takes a mostly hardware-based approach in its TEE design. This should come as no surprise, seeing how Intel remains one of the largest CPU manufacturers in the world.

Components Intel slightly adapts the original TEE architecture by inverting the relationship between container and code. Instead of providing a single container where new applications are executed, SGX can provision an arbitrary number of secure environments as part of an already running application. In the context of SGX, these environments are referred to as *enclaves*. They are located within a protected section of the application's memory, where they provide strong confidentiality and integrity guarantees for the contained code and data.

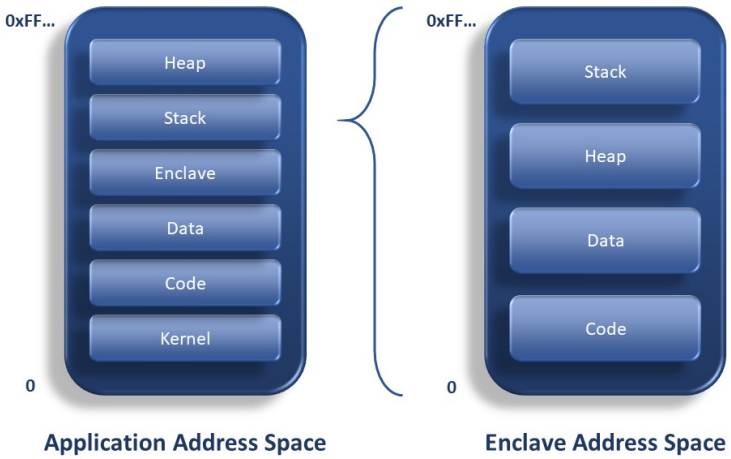
Enclave functionality is enabled by specialized microprocessors implanted on newer generations of Intel's mainline CPUs [72]. A high-security CPU mode with dedicated instructions is required to read or write an enclave's memory. It may not be accessed otherwise, not even by privileged software, such as the OS or the BIOS.

To convince others of their correct operation, enclaves can issue attestations. Attestations allow third parties to remotely verify that an enclave was setup up correctly and that it is still running the original code. Enclaves also provide a technique called *sealing*. Sealing refers to the act of securely exporting enclave data in encrypted form. It can be used to share secrets with other enclaves or reclaim enclave memory by offloading data.

Memory Model SGX enclaves support modern CPU memory management techniques like memory paging and virtual addressing. As a result, an enclave can be fully contained within its respective application's virtual address space. Unlike the rest of the application's memory, however, enclave memory pages cannot simply be swapped in and out by the OS. Storing the pages on disk could potentially leak sensitive information to a malicious host, severely compromising the enclave's security in the process. Hence, additional steps are required to protect them.

This role is fulfilled by the *Enclave Page Cache (EPC)*, an encrypted region of memory that serves as a container for enclave pages. The EPC is contained within an access-restricted form of memory called *Processor Reserved Memory (PRM)*. It is allocated by the BIOS during boot and may only be accessed through specialized CPU instructions. Once the EPC has

Figure 4.1.: Memory layout of an initialized SGX enclave.



been instantiated, an enclave can store its sensitive memory pages and other control structures in it. Through virtual addressing, these pages can then be loaded into the executing application's address space.

Since all enclaves share the same EPC, special security measures are necessary to ensure that individual pages can only be accessed by their owning enclave. Such page permissions are tracked by the *Enclave Page Cache Map (EPCM)*. This simple access-control list contains detailed information for every single enclave page on the machine, e.g. owning enclave, page type, address, etc. Whenever a page is requested, the processor first consults the EPCM if the requesting enclave is indeed the owner of the page. As is the case with the EPC, access to the EPCM is also highly restricted. Only the *Page Miss Handler (PMH)* – a dedicated hardware chip on the CPU – can read information from the list.

Besides providing a protected environment for enclave memory, the EPC is also responsible for securely swapping memory pages in and out from disk. Before an enclave page can be evicted, the EPC first labels the page with a unique version number and saves this number in a control structure. The version number can later be used to verify the page's freshness when loading it back into memory. By prohibiting the load of outdated pages in this way, replay attacks can be prevented. After a page has been labeled, the EPC encrypts it and finally writes it out to disk. Additionally, any buffers and caches containing pointers to the page must be cleared. Especially the *Translation Lookaside Buffer (TLB)*, present on many modern CPUs, contains sensitive data that can be used by an adversary to partially reconstruct a page's contents.

4.2.3. Operation

There are two main functions provided by SGX: *secure program execution* and *remote attestation*. We will go over each of them individually.

Secure Program Execution To provision a new enclave, the CPU must first create an *SGX Enclave Control Structure (SECS)*. This data structure holds all enclave-related metadata, most importantly the enclave's multiple identities. Various initialization parameters are also stored in the SECS. These include the address range of the memory pages, the enclave's operating mode, as well as its supported features. Once the SECS has been initialized, more free pages in the EPC are marked as enclave memory and registered with the EPCM. From now on, only the newly created enclave has access to these pages.

After allocating the enclave memory, the CPU can now load the application's code into the pages. Because the code is loaded from within the untrusted host's environment, it must be protected against tampering. To do so, a dedicated CPU instruction "measures" the initialized pages in 256 byte steps, i.d. a cryptographic hash of the relevant memory region is computed and stored in an enclave-specific append-only log.

When all pages have been measured, the log now contains a unique fingerprint of the entire enclave's memory. The hash of this log is called the *Enclave Identity* and is stored in the identity register of the SECS. This Enclave Identity uniquely identifies the enclave based on its contents. It can also be used by a third party to verify that the application code was loaded correctly by the untrusted system.

Another identity, the *Sealing Identity*, is also stored in the SECS. This second identity refers to the underlying platform, which built the enclave. It is therefore not unique and can be used to encrypt and share data across enclaves running on the same system. The Sealing Identity is constructed from several CPU parameters and the enclave builder's public sealing key. During enclave initialization, the builder injects its Sealing Identity into the enclave.

Before it can be stored in the SECS, however, the sealing key must be verified. To this end, the builder pre-computes the Enclave Identity and signs it using the private portion of the sealing key. Since only the builder has access to the memory pages during the enclave creation process, a correctly computed and signed Enclave Identity proves that the sealing key is genuine.

Storing the Enclave and Sealing Identity in the SECS concludes the enclave initialization process. The CPU can now run the enclave's code just like regular application code by switching to and from the enclave CPU mode. Unfortunately, constant switching between the regular and secure mode of operation opens the enclave up to another set of attack vectors. Because modern CPUs feature a host of different registers and caches, a malicious adversary could read leftover sensitive information from these storage locations, even after a context switch has occurred. As is the case with paging, all buffers must therefore be cleared before the CPU context can be switched.

For planned enclave entries and exits, this process happens synchronously and can be hardwired into the application code. Enclave exits caused by program exceptions or I/O-interrupts, on the other hand, cannot be foreseen by the compiler. In both cases, the CPU executes an *Asynchronous Enclave Exit (AEX)*. Execution is trapped in a special handler function that extracts the sensitive information from the various registers and encrypts them within the enclave. Subsequently, the registers are filled with fake values, preventing reconstructions of the enclave state. When execution flow re-enters enclave mode, the saved values are decrypted and restored to the CPU's registers.

Enclave Attestation While the enclaves themselves might be trusted, they are initialized by an untrusted system. Because of SGX's strong confidentiality guarantees, it is impossible for an observer to see whether an enclave was set up correctly. To overcome this limitation, SGX provides a comprehensive attestation system, that is able to convince third parties that a running enclave is indeed trustworthy [6].

SGX attestations come in two forms. Internal attestations allow enclaves running on the same platform to authenticate one another. Developers can use this feature for secure inter-enclave communication, allowing them to create programs spanning multiple enclaves. Remote attestation builds on top of the internal attestation infrastructure by additionally signing the attestation document. This way, any third party located outside the enclave can verify the attestation using a special Web service hosted by Intel.

Internal attestation revolves around the exchange of so-called "reports". These digital documents are generated by the attesting enclaves to prove their trustworthiness. A report must contain the two enclave identities, several initialization parameters, and a trustworthiness measurement of the underlying system. It may also carry optional user data. To prove that a report is genuine, the attesting enclave appends it with a cryptographic MAC.

The MAC's key is not known to the enclave. Instead it is generated confidentially with the help of a dedicated CPU instruction. An enclave receiving the authenticated report can obtain the secret MAC key by using another one of SGX's specialized instructions. A valid MAC means that the report was constructed within the same SGX unit. The Enclave Identity can then be extracted from the report to verify the enclave's contents.

The remote attestation process, on the other hand, resembles a traditional challenge-response protocol. To begin, a third party (verifier) creates the challenge and adds a nonce providing freshness. The challenge is then relayed to the node running the attesting enclave (prover). Before the challenge can be processed further, the prover must first identify its *Quoting Enclave*. This special enclave is always active on SGX-enabled systems. Its sole purpose is to act as a signature provider for remote attestations. Once the prover has consulted the Quoting Enclave, its ID along with the original challenge are passed to the application enclave, whose trustworthiness must be established.

The application enclave then proceeds by creating the attestation manifest. This data structure is only necessary for remote attestation and will be issued to the verifier. It contains a response to the original challenge and an ephemeral encryption key. This key can be used to establish a secure communication channel between prover and verifier after the protocol has completed. Additionally, an internal attestation report is constructed. Besides the usual information, this report also contains the manifest's hash in its optional data slot. The completed report is now forwarded to the local Quoting Enclave.

If the Quoting Enclave deems the report valid, it replaces the report's MAC with a device-specific signature. The signing key is derived from the platform's hardware using Intel's *Enhanced Privacy Identifier (EPID)* signature scheme.[1] Once signed, the Quoting Enclave hands the modified report back to its host, which then returns it to the verifier. The verifier checks the signature using either a public certificate or Intel's public attestation service. If the signature is genuine, the manifest can be safely extracted from the report. The remote attestation was successful, if the obtained manifest contains a valid

[1] EPID is a group signature scheme, which guarantees privacy by ensuring that signing keys cannot be used to track individual SGX devices. For a more detailed description, the reader is referred to [59].

response to the issued challenge. In this case, the verifier can rest assured that the enclave was indeed initialized with the correct parameters and is still running its original code.

5. Implementations

The need for trusted oracles has been at the forefront of blockchain development since the earliest days of smart contracts. Consequently, a substantial number of different implementations exist. In the following, we highlight some of the most important players and explain their inner workings in detail.

5.1. Overview

In accordance with the technologies presented above, most oracle trust mechanisms can be classified into either game-theoretic or cryptographic. We will now list three actively developed examples for both categories. In the interest of brevity, we restrict our selection to oracle service that are specifically intended for verifiable computation.

Game-Theoretic Oracles The following oracle systems rely on long-standing observations from game theory. All of them are implemented using some form of incentive-compatible reward scheme.

- TrueBit [92]

  One of the first verifiable computation oracles and fully compatible with Ethereum smart contracts. Users submit their computations to the service accompanied by a bounty of corresponding size. A single node is selected as the solver at random. Assuming an honest solver, the node then runs the assigned computation and publishes its result to the blockchain. Verifiers now have the chance to re-execute the computation and challenge an incorrect result, earning them a reward.

  To incentivize verifier participation, TrueBit's forced error protocol ensures that a sufficient number of incorrect solutions exist. As a part of this protocol, every solver is required to prepare an additional, erroneous computation result. This second result must differ from the previously calculated correct one. A randomized algorithm then chooses which of the two results must be published. Naturally, solvers who publish an incorrect result in this way are not penalized for it.

  When a result is challenged, TrueBit executes a "verification game" to establish if the challenge is justified. To this end, the solver and the verifier engage in an interactive binary search of the execution path to determine the exact point where their computations begin to diverge. Once the offending step has been located, the correct result is determined by letting the Ethereum network re-execute it. The performance penalty incurred by this is negligible, since it happens rarely enough and it only applies to a single execution step, not the whole computation.

- Astraea [1]

  A fully incentive-driven oracle network that is realized using only on-chain components. Built on the Ethereum blockchain, it answers client requests posed as "Boolean propositions", i.e. simple yes/no questions. Oracles vote on these propositions by supplying whether they think the proposed statement is true or false.

  To incentivize participation, nodes can sign up to become one of two different kinds of oracles, depending on the risk they are willing to take. *Voters* only place a small security deposit and are assigned a random request. They cast their vote and receive a small portion of the request fee if voted correctly. *Certifiers* are allowed to choose the proposition they vote on, but are required to place a larger stake than the voters. However, by choosing correctly, they also stand to gain a much larger reward. Additionally, if their vote turns out to be false, certifiers are forced to pay a fine to other certifiers who voted truthfully.

  Originally intended for voting on future events in the style of prediction markets, Astraea's oracle scheme can be adapted to solve off-chain computations. To do so, an initializer first solves the computation off-chain and asks the network if the result is correct. Now the other oracle nodes solve the computation themselves and confirm or reject the result. Assuming an honest majority, the correct result will be revealed after several rounds.

- Chainlink [42]

  A decentralized oracle designed with the *Oracle-as-a-Service (OaaS)* paradigm in mind. Its most characteristic feature is a publicly available, continuously updated reputation ledger for participating oracle nodes. The individual reputation scores serve as a foundation for Chainlink's own monetary incentive scheme, including reward and penalty payments. They further enable users to make informed decisions when choosing which oracles to trust.

  Chainlink is built on Ethereum and uses the native ERC20/223[93, 38] currency *LINK* for all internal transactions, e.g. oracle fees. In its original incarnation, the service was intended as a simple means for on-chaining data from Web-APIs. Chainlink's modular architecture, however, allows users and oracle providers to execute their own data processing scripts, greatly expanding the number of possible use-cases.

  More recently, Chainlink has also started to move to a hybrid security approach, that includes trusted hardware in addition to its existing incentive scheme. With the help of TownCrier's TEE infrastructure (see below), the Chainlink developers plan to support fully encrypted, remotely attestable off-chain computations in the future.

Cryptographic Oracles The following oracles are powered by the cryptographic capabilities of TEEs. Note that all implementations in this section are built on top of Intel SGX (conf. 4.2). This is simply due to a lack of oracles with alternative underlying TEEs.

- TownCrier [102]

  An oracle service that leverages SGX to securely on-chain data from Web-APIs. Its public interface is available as a smart contract hosted on the Ethereum network.

Client contracts can submit queries containing an endpoint, a callback function, and a collection of API parameters.

When TownCrier's smart contract receives a query, it is forwarded to one of the oracle nodes (AKA "relay servers") through Ethereum's event system. Using a local SGX enclave, the relay servers execute the requested API call in a trusted environment. When the computation completes, its result is returned to TownCrier and from there to the callback function in the client's contract. To ensure that the API was accessed properly, the relay servers create an attestation of the enclave and include it in their response. This attestation can easily be verified by the client contract.

Moreover, TownCrier supports the use of confidential requests. Using the enclave's public key, a client can encrypt their query's payload, making it only accessible from within the trusted environment. Allowing encrypted queries in this way not only protects privacy but also lets users authorize to protected APIs using their personal credentials.

- Kosto [36]

  A market place for verifiable computations on Ethereum with a strong emphasis on fairness. It is maintained by several independent broker nodes. The brokers load balance computational work by establishing new connections between clients and compute nodes according to a bipartite matching algorithm. This algorithm takes into account various parameters, e.g. computation size, reward amount, payment details, etc.

  When a match occurs, the client deposits a reward fee and submits the program they wish to outsource. Its code is compiled and further enhanced with "dynamic runtime checks". These are necessary for the fine-grained payment scheme with which Kosto pays its compute nodes. The reward fee is split in a computation and a delivery portion. The delivery portion resembles a classical bounty for calculating the correct result. Unlike other platforms, however, Kosto also rewards nodes for the mere act of computing. As such, the second portion of the reward is paid out incrementally over a micro-payment channel for every runtime check the compute node passes.

  The verification of completed computations also happens via the brokers. Once a program has been compiled, it is loaded into an SGX enclave on the chosen compute node. An attestation of the application enclave is created and verified by the broker. The verification signature is then cached locally in the broker's enclave for later retrieval by the clients. This removes the need for every client to contact Intel's attestation service, greatly reducing the required bandwidth.

- Ekiden [27]

  A smart contract development platform replicating the virtual state machine of Ethereum, but with significant privacy and performance improvements. The project is still mostly academic with no production-ready build yet, but a thoroughly tested reference implementation is described in the original white paper. The reference implementation was built on top of Tendermint [69], but Ekiden itself is blockchain agnostic, meaning that any blockchain can be used for the underlying storage model.

Name	Whitepaper	Technology	Request Model	Production-Ready
Astraea	[1]	IOC	Boolean Propositions	No
TrueBit	[92]	IOC	General Purpose	Yes
ChainLink	[42]	IOC+SGX	Web Requests (General Purpose tbd.)	Yes
TownCrier	[102]	SGX	Web Requests	Yes
Kosto	[36]	SGX	General Purpose (Single-Threaded)	No
Ekiden	[27]	SGX	Smart Contracts	No

Clients formulate their requests using "private smart contracts", small pieces of code that can be written in a number of well-known programming languages. Once deployed, these contracts can be called with a combination of public and encrypted private inputs. Despite also being based on blockchain technology, Ekiden achieves better performance by reducing the number of interactions with this relatively slow computation environment. This is done by moving computations off the chain and onto oracle nodes; only the resulting state updates are persisted on the blockchain. To prevent malicious oracles from tampering with the contract or the input data, Ekiden also relies on Intel SGX for secure program execution.

5.2. Deep-Dive: Chainlink

We now take a closer look at the inner workings of Chainlink [42]. Several factors influenced us in our choice to study Chainlink in greater detail: its active community, its extensible architecture and most importantly the presence of production-ready software.

5.2.1. Security Concept

Chainlink's security claim rests on the foundations of incentive-compatible distributed networks. The software makes use of this in two different places:

1. A single Chainlink oracle may retrieve its data from more than one trusted source. The final result is obtained by feeding all data points into an aggregation function. During this intermediate step, oracle operators have an opportunity to do additional data processing to ensure optimal results. Such tasks may include the filtering of outliers, smoothing of fluctuating data, or advanced error handling. The function itself is customizable and may be adjusted to handle different data types.

2. A user's query is always answered by multiple oracle nodes. Before the final result is returned, a second aggregation step occurs that collects all oracle answers and calculates the final value. The exact method is supplied by the user when submitting their query and can be customized just like in the first aggregation step. Examples include averaging the values, only accepting the majority result, etc.

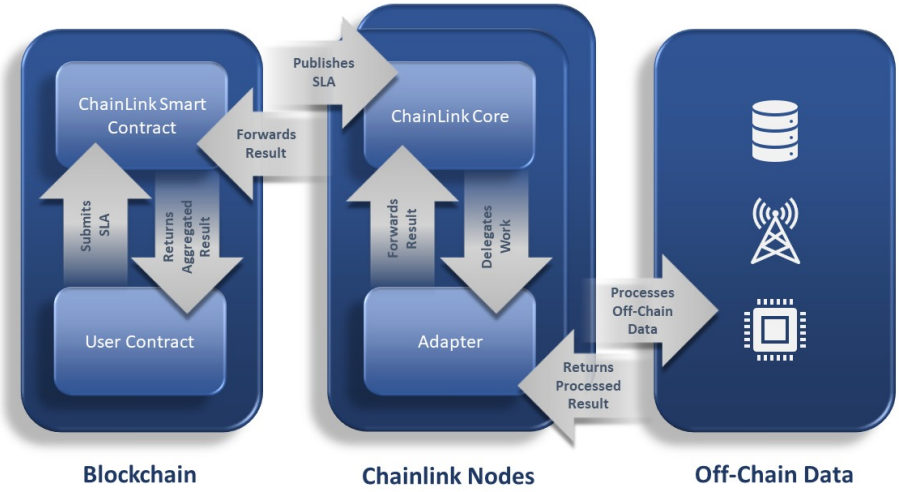


Figure 5.1.: Schematic overview of Chainlink components.

Whereas the initial aggregation step exists to help the oracle providers sanitize their raw data, the second step acts as a correctness check for the oracle answers. To ensure transparency and traceability, this second aggregation is executed as a smart contract that is part of Chainlink itself. This smart contract is also responsible for managing the incentive scheme. All oracle nodes receive monetary rewards or punishments based on whether their answers were accepted by the user. These outcomes are permanently recorded in a public reputation ledger. Chainlink's security relies on the assumption that oracles behave honestly to maintain a high reputation, thus maximizing their potential rewards.

5.2.2. Architecture

Chainlink relies on a number of different interlocking parts to do its job. These parts can be split into two distinct categories: on- and off-chain.

On-Chain The on-chain infrastructure is comprised of three Ethereum smart contracts that stand in close relation to one another. We take a look at each one individually.

- Reputation Contract

  The reputation score for each oracle is stored publicly in the reputation contract. A given score is calculated from a number of performance metrics collected during the second aggregation step. For each oracle the number of total requests, the number of total responses, and the number of those responses which were accepted by the user are logged. The ratio of requests to accepted answers yields the main portion of the score. Further adjustments are made based on the oracle's average response time, as well as the total amount of penalty payments made over time.

The resulting reputation ledger serves as a guideline for future users when selecting which oracles should handle a specific request. Users may either access the scores directly to manually search for oracles or let the process be automated by the order-matching contract.

- Order-Matching Contract

  The order-matching contract is Chainlink's main user interface. It implements a decentralized auction platform where users can submit requests and oracles may bid on their fulfillment. The matching logic respects boundary conditions embedded in the requests. Besides general reputation requirements, users may specify a variety of other conditions, such as the minimum number of answers before a result is finalized. Reputation information is automatically pulled from the reputation ledger before the start of each bidding period. If an oracle's score is lower than what the user demands, any bids will be rejected by the matching algorithm.

- Aggregating Contract

  The aggregating contracts serves as a central interface to which oracles can report their answers. Once the desired number of answers for a request is reached, the aggregating contract forwards their values to the chosen aggregation function. Finally, the aggregated result is returned to the user and reputation scores adjusted accordingly.

  The aggregation function itself is not part of the aggregating contract, but is instead outsourced to another smart contract. This two-step design is necessary because each request works on different kinds of data. Considering the sheer number of options, the Chainlink developers are thus unable to provide a single aggregation function that is compatible with all possible data types. Chainlink already comes with contracts for some of the most common operations, but also allows users to implement their own aggregation functions. To specify a custom aggregation function, a user simply includes its contract address in their initial request.

An important implementation detail that deserves mentioning, is Chainlink's protection against *freeloading attacks*. Because oracle answers are submitted in the clear, a malicious provider could simply copy the most common answer and report it. As such, the attacker receives a reward without doing any actual work. To prevent this, the aggregating contract uses a commit-and-reveal scheme for submitting answers. At first, oracles merely supply a hash combining their identities and answers. Upon request finalization, the cleartext values are revealed and replace the corresponding hashes.

Off-Chain The off-chain components are entirely located on the oracle nodes. They include the following:

- Chainlink Core

  The official Chainlink client which facilitates communication between the on- and off-chain environments. It serves as a direct bridge between the order-matching and the aggregating contract on the one side, and request execution on the other. Incoming requests are read from the Ethereum event log and ecapsulated in a standardized

JSON format. Outgoing answers are serialized and inserted back into the chain for result aggregation. For the actual data processing itself, request objects are passed on to *Adapters*, scriptable software modules unique to each task.

- Adapters

  Small data-processing scripts running in parallel to Chainlink Core. As is the case with outsourced aggregation functions, this design aims to cover as many use-cases as possible without unnecessarily inflating the core's code size. Adapters are written in plain JavaScript. Aside from passing JSON objects to and from the core software, they contain the main interaction logic, ranging from simple API calls to parsing jobs and even inter-chain transfer of user data. Chainlink Core includes a number of common oracle tasks as adapters, but oracle operators are free to implement any arbitrary off-chain computation with the provided JavaScript libraries.

5.2.3. Operation

From an end user perspective, Chainlink is intended to work as a black box device for on-chaining data. This is in line with the idea of Oracles-as-a-Service. We proceed to outline the general usage for someone wanting to procure such a service.

Request Lifecycle User requests must be submitted as *service-level-agreements (SLA)* to the order-matching contract. This simple data transfer object encapsulates all the details of a single transaction. Besides the contents of the request itself, an SLA can contain several query parameters which modify the way a request is handled. This includes the required number of oracles, their minimum reputation score, and many other boundary conditions. The finalized SLA is sent together with a reward fee for the participating oracles.

Once the order-matching contract has received the request, an Ethereum event is triggered, informing the oracle nodes of the new listing. The Chainlink Core software running on the oracle nodes picks up the event and, depending on the provider's preference settings, places a corresponding bid. If the bid is accepted by the order-matching contract, a penalty payment is deducted in case the oracle answers incorrectly. The bidding period ends as specified by the user, either when the required number of oracles have placed a bid, or the request timed out.

Oracles chosen by the matching algorithm receive a notification and start processing the request data. They then feed the results returned by their adapters back into the aggregating contract. Through the supplied aggregation function, the various answers are unified into a single final result. With each accepted or rejected answer the performance metrics are reevaluated and reputation scores adjusted accordingly. Finally, the oracles deemed correct receive their penalty deposits and in addition a reward payment taken from the initial request fee. The final result is returned to the user and a request's lifecycle completes.

Use of Trusted Hardware One of Chainlink's long-term goals outlined in the original white paper is the use of trusted hardware to supplement the existing incentive-driven security approach. To this end, the developers have partnered with the TownCrier project

to introduce TEEs to Chainlink's back end infrastructure. In practice, this means using the same Intel SGX devices already present in TownCrier's architecture. The plan is to deploy the TEEs on the oracle nodes as a secure environment to run their adapters. Two additional security guarantees could be gained by introducing trusted hardware.

Firstly, since every TEE possesses its on public encryption key, users would have the option to encrypt any sensitive data contained in a request. This way, the data can only be decrypted from within the TEE by the currently running adapter. Consequently, no malicious oracle would have access to the private information and since a user can verify a TEE's loaded code, any malicious adapter is quickly discovered. Thus, by encrypting private information, confidentiality can be achieved. This allows the inclusion of sensitive data, such as user passwords, identity information, etc., significantly broadening the available use-case spectrum. Public key encryption of requests is supported by TownCrier out-of-the-box in the shape of custom datagrams [102].

Secondly, it is the Chainlink team's declared goal to introduce fully secure off-chain computations with the help of TEEs. In this model, oracle answers are computed directly within an enclave and must not rely on external resources, such as public APIs. Being able to remotely attest the executed code endows users with a much stronger notion of trust. It should be noted that, since no external data can be retrieved from within an enclave, such off-chain computations only make sense if all required data is known in advance.

5.3. Deep-Dive: Ekiden

What follows is an in-depth description of the Ekiden smart contract platform. We chose to focus on Ekiden because it offers strong privacy guarantees and a familiar programming environment for developers.

5.3.1. Architecture

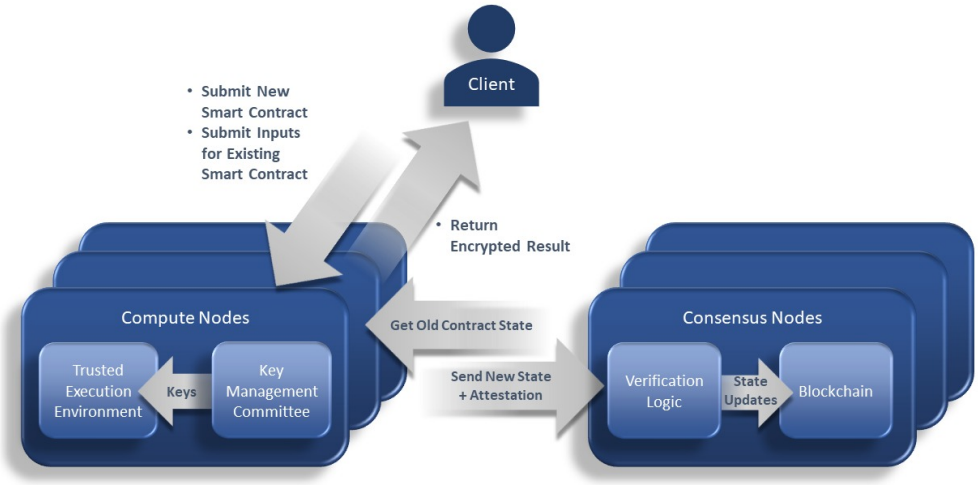
Ekiden's architecture revolves around three distinct types of network nodes. Each type is responsible for a different support function of the platform. They are:

- Client Nodes

  Users who submit their computations to the oracles are referred to as client nodes. They are usually small, privately-owned devices with low performance, such as cellphones or web browsers. The computations themselves are specified as smart contracts. Clients can call existing contracts or create a new computation by publishing their own. This is currently done using either a subset of the Rust programming language or Ethereum's own smart contract language, Solidity.

  Unlike traditional smart contracts, Ekiden supports the use of private inputs. Private inputs serve as arguments to a contract's execution, but remain invisible to outside observers during runtime. Also, unlike the Ethereum Virtual Machine, Ekiden does not execute its smart contracts redundantly on all nodes. Instead, they are delegated to a small network of compute nodes for better performance. The compute nodes are equipped with TEEs to rule out malicious behavior.

Figure 5.2.: Information flow diagram of a single Ekiden instance.



- Compute Nodes

  Any machine containing an Intel SGX unit can sign up to become a compute node for Ekiden. It is a compute node's job to accept client requests and return a correct result by executing the respective smart contract.

  Besides yielding a computation result, an Ekiden smart contract may contain persistent state information that is accessed and changed while handling a request. Because this state information is consistently updated, the compute nodes effectively run a distributed state machine similar to the Ethereum EVM. Unlike the Ethereum EVM, however, smart contract execution is only ever delegated to a single compute node. This leads to much better performance when compared to the traditional model of redundant blockchain execution. Once completed, the individual state updates are forwarded to Ekiden's consensus layer where they are persisted between contract executions.

  Because Ekiden's state machine lacks a distributed consensus algorithm that ensures correct contract execution, this responsibility falls on the TEEs located within the compute nodes. First of all, the TEEs provide confidentiality by encrypting each and every state update before placing it on the blockchain. Secondly, they use their built-in cryptographic mechanisms to generate publicly verifiable attestations for their computations. The attestations claim that the contract was executed correctly and that the resulting state is really what was encrypted. By consulting Intel's attestation service, any third-party can verify their validity, event without the use of trusted hardware.

- Consensus Nodes

  Ekiden is designed so that state updates can be persisted using any current blockchain implementation. In fact, the nodes making up the consensus layer are simply ma-

chines operating a well-known blockchain protocol. Only a thin layer of additional verification logic is required to check whether the attestations created by the compute nodes are legitimate. Whenever a state update is received by a consensus node, it verifies the attestation's correctness and appends it to the underlying blockchain along with an encrypted update of the contract state.

5.3.2. TEE failure mitigation

While TEEs provide numerous strong security guarantees, they are not infallible. Because of their restricted computing environment, TEEs cannot offer all of the same services as conventional CPUs. This makes them reliant on untrusted outside sources for some operations. Recently, due to their increasing popularity, TEEs have also become the target of concentrated hacking efforts. Numerous attacks on TEEs, in particular Intel SGX, have already been discussed in literature (conf. 13.1). Ekiden acknowledges these risks and introduces a collection of new security protocols in response.

Proof-of-Publication Intel SGX does not feature a trusted absolute time source. An internal timer can keep track of the elapsed time since a given reference point, but has no access to the current date and time. To access time related functions, an SGX enclave must refer to the underlying OS for help. Even though the timestamp received this way is cryptographically signed, it can still be delayed by a malicious host.

This becomes a problem whenever the enclave wants to read an item from the blockchain. Without an accurate trusted time source, an attacker is able to stage an isolation attack, as described in [29]. During an isolation attack, the enclave sees a forged side-chain created by the attacker. By falsely including specific messages in the fake chain, an attacker can try to convince the enclave of a certain event, when in reality the offending messages were never published to the rest of the network.

Ekiden limits the success rate of isolation attacks by employing a *proof-of-publication*. This technique allows any enclave to verify whether a message was really published to the main chain. Depending on the chosen parameters, the chance for a fraudulent message to be accepted becomes negligible. Once a proof-of-publication for a certain message exists, other compute nodes can verify it without having to re-execute the protocol.

Proofs-of-publication are implemented as interactive proofs between an enclave (verifier) and a message sender (prover). The enclave begins a proof by sampling the average block creation time on the main chain. Then the enclave requests a first timestamp t_1 from the OS and, upon receipt, sends a random nonce to the prover. The prover now has a limited amount of time to publish a block containing their message as well as the random nonce. Once the enclave observes such a block in the chain, it creates a second relative timestamp t_2.

Looking at the number of confirmation blocks on top of the original message, the enclave can safely estimate how much time it would have taken the honest network to build such a chain, since it already knows the average block interval. By comparing this number to the difference between the sampled timestamps, it is possible to gauge if an isolation attack is taking place. Unless the attacker is in possession of a majority of the blockchain's mining power, the fastest way to mine a block is behaving honestly and sharing the message with

the whole network. Therefore, if the measured time is close to the time it took to create the chain, it is probable that the chain was mined by honest nodes.

Key Management Committees Operating Ekiden's various protocols requires the use of cryptographic keys. It seems practical to store these keys in the secure environments spawned by the Intel SGX units. Unfortunately, recent studies have shown that SGX and many other TEEs suffer from vulnerabilities to side-channel attacks given a powerful-enough adversary (conf. 13.1).

Compromising a long-lived encryption key would be devastating for user privacy. To resolve this, several compute nodes band together to form a *key management committee*. Using advanced cryptographic techniques, these nodes derive and distribute encryption keys to authorized parties. As a result, the encryption keys are secure unless an attacker can subvert all nodes that are part of the committee. This feat seems improbable given the currently known side-channel attacks on SGX.

Ekiden keys in are classified into two categories: long- and short-term keys. A single long-term key is generated by the key management committee. Since storing the key directly in the enclaves would be insecure, its nodes engage in a distributed key generation protocol [45] to split the task. At no point during the protocol does a node gain knowledge of the full key. Instead, each participant receives a key fragment that can only be reassembled when all other fragments are present. Therefore, the key cannot be restored unless all nodes collaborate. This makes it increasingly hard for an attacker to leak the key, as all nodes would need to be compromised.

It is considered a best-practice in modern key management systems, to not use the long-term key directly for data encryption. Rather, it serves as a seed to generate a fresh short-term key for every new encryption request. This way, if a short-term key is ever compromised, e.g. through a side-channel attack, only the data encrypted with that particular key can be stolen. Ekiden calls this property *forward secrecy*[1].

A contract TEE may request a short-term key by engaging in a distributed pseudo-random function protocol [78] with the key management committee. More specifically, the committee members each use their key fragment as an input to a randomized key generation function. They learn neither the other members' fragments nor the function's final output. The resulting short-term key only becomes known to the enclave that initially posted the request.

Atomic Delivery Protocol Successful execution of a Ekiden smart contract yields a computation result and an encrypted state update. The requesting client receives the computation result, whereas the new contract state is forwarded to the consensus nodes. Because these messages are transmitted independently and over insecure channels, the chance of either one becoming lost in transit is reasonably high. A skilled enough adversary might even cause this on purpose, e.g. with a targeted DoS attack. This can result in several problems for Ekiden's state machine.

If the updated contract state is appended to the blockchain, but the output never arrives at the client, it becomes irrevocably lost. Depending on the nature of the contract, it might

[1] The conventional definition of forward secrecy also requires that compromised long-term keys do not affect the security of the short-term keys.

also now be impossible to reproduce the original output, as the new state was already persisted. This technique serves as a powerful tool for DoS attacks against specific targets.

In the case of losing only the state update, the resulting data inconsistency can be leveraged by an attacker to execute a *rewind attack* against the system. Rewind attacks are possible, if the executing smart contract contains a randomized algorithm, such as a public lottery. To do so, the attacker simply acts as the client and runs the contract. If the result is not favorable, the contract can simply be re-executed from its previous state until the favorable outcome is achieved.

To limit the chances of successfully executing such attacks, Ekiden employs its own *atomic delivery protocol*. This ensures that both items are either delivered or dropped, but never one without the other. First, the compute node generates a symmetric encryption key and sends its computation result to the client, but only in encrypted form. Then, the compute node publishes the state update to a consensus node and waits for its inclusion in the blockchain. This can be ensured with a proof-of-publication (see above). Once the state was included in the chain and a proof has been received, the compute node finally sends the required key to the client. The client can now use this key to decrypt the original message and view the computation result.

5.3.3. Operation

The Ekiden platform provides two major functions for its users. They can either create blueprints for new computations by submitting a new contract or execute existing computations by calling a contract.

Contract Creation To implement a smart contract, a client first needs to write its code using one of the available languages and then upload it to a compute node. From there, the code is loaded into the compute node's enclave. Next, the enclave queries the key management committee for a fresh key suite consisting of a symmetric encryption key and an asymmetric key pair. Contract states are encrypted with the symmetric key, private client inputs using the public key of the asymmetric pair.

At the start of the contract creation process, the enclave also encrypts the uninitialized contract state. This serves as a common reference point for later executions. To prove that the correct state was encrypted, the enclave creates an attestation for it. Now, the public encryption key, the contract's code, its encrypted state and the attestation are all passed on to a consensus node.

Once that data is received, the consensus node verifies that the attestation is correct and, if the check passes, places code, public key and encrypted state on the blockchain. This information can be used by other nodes to execute the contract and advance it from one state to the next.

Contract Execution When a client wants to execute a contract, they must first obtain its public key from the blockchain. With this public key it is possible to encrypt private inputs and send them to the executing enclave in secret, without the danger of a spying host. After receiving the client's private inputs, the enclave retrieves the contract's code and its current state from the blockchain. It also asks the key management committee for the

corresponding keys to decrypt the contract state as well as the private inputs. Once this data has been decrypted, the requested computation is ready to be executed in a secure environment.

Two artifacts are obtained after executing the smart contract. They are transmitted via Ekiden's atomic delivery protocol. The actual output gets sent directly to the client node. The contract's new state is encrypted and combined with another attestation, closely resembling the setup described during contract creation. A message containing the encrypted state as well as its attestation is sent to a consensus node. If the attestation can be verified, the new state is securely embedded in the blockchain. This completes the user's request and advances the Ekiden virtual machine by one step.

Part III.

Zero-Knowledge Proofs

6. Concept

Can we proof knowledge of a fact without revealing it? – This is the fundamental problem which zero-knowledge proofs seek to solve. Interestingly enough, the very concept of this technology dates all the way back to the 80s, where Goldwasser et al. [49] gave the following definition: A zero-knowledge proof

1. must convince the verifier that the prover indeed knows the fact,
2. may not be forged by a malicious prover without knowledge of the fact,
3. may not allow the verifier to obtain knowledge of the fact.

Informally, these three properties have become known as *Completeness*, *Soundness* and *Zero-Knowledge*, respectively.

The Strange Cave of Alibaba [85] is a famous literature example which illustrates these different characteristics. Prover P and Verifier V stand in front of a ring shaped cave with one entrance. At the far end of the ring, a magic door blocks the path and opens only to people who utter the correct passphrase. P wants to convince V that they know the phrase without revealing it. To achieve this, V turns around and P enters the cave either to the left or to the right. Without knowledge of P's chosen path, V turns to face the cave again and shouts the direction from which P should emerge. P returns on the chosen path, opening the magic door if necessary.

By repeating this process a sufficient number of times, we obtain a protocol which satisfies the definition of a zero-knowledge proof.

1. Completeness

   If P emerges from the correct path every time, V will eventually be convinced that P knows the phrase.

2. Soundness

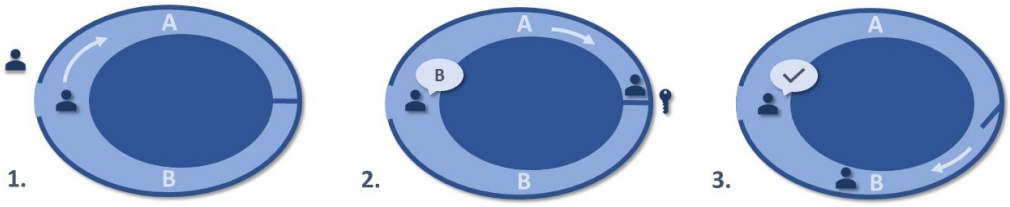
   If P does not know the phrase they are unable to open the door. The probability of choosing the same path as V halves every time and eventually becomes negligible.

3. Zero-Knowledge

   Trivially, V does not learn the phrase at any point during the protocol.

After a period of relative silence around the state of zero-knowledge proofs, their development has recently regained traction with the introduction of *zkSNARKS* [44]. This new class of zero-knowledge proof supports additional functionality that is highly desirable for blockchain applications. Initially employed by Zcash [55] to hide transaction details for their cryptocurrency, zkSNARKs find increasingly widespread use in other blockchain projects.

Figure 6.1.: High-level depiction of the *Alibaba's Cave* illustrative example.



Arguably, their most promising application is the planned introduction of verifiable computing in Ethereum 2.0. As a foundation for the so-called "ZK-Rollups" [1], the Ethereum developers hope to vastly increase their transaction throughput by placing expensive computations outside the chain. In the following, we take up this technique for our own verifiable computation approach.

[1] https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/

7. Background

As mentioned in 6, zero-knowledge proofs have experienced a recent resurgence in the blockchain space with the invention of zkSNARKs. We take a look at how these novel proving systems work as well as the mathematical principles which form their foundation.

7.1. zkSNARKs

zkSNARKs are specifically designed as a cryptographic primitive for verifiable computation. They combine state-of-the-art correctness arguments for mathematical functions with traditional zero-knowledge techniques. This grants users not only the ability to publicly verify arbitrary computations but also lets the provers hide any secret values that were used during said computation. Despite being a relatively young technology, great advancements in the realm of zkSNARKs have been made within the last decade.

Formal Definition The term SNARK was coined by the authors of [19], when they hypothesize the existence of mathematical proofs for arbitrary computational statements of the form $y = f(x)$. It stands for **S**uccinct **N**on-interactive **AR**gument of **K**nowledge. This acronym expands on the initial definition of zero-knowledge proofs as follows:

- Succinct

  The proofs generated by a SNARK are short. More specifically, their length must be polynomially bounded by either the computation's output size or the chosen security parameter k. This property is especially important for blockchain applications due to their limited amount of storage.

- Non-interactive

  Proof verification happens independently of the prover. Once a proof has been published, the verifier can check its integrity with an isolated computation and does not need to engage in a two-way protocol with the prover. This is usually done by simulating such a two-way protocol with a randomized proof, where the randomness bits are be chosen by the verifier in advance. Whether the correct randomization was applied can then be checked offline with the corresponding verification key.

  The verification key is not bound to a single verifier and can be shared among multiple parties. This makes the proof publicly verifiable by anyone in possession of the key. Such public auditability is another important condition for the effective use of zkSNARKs in blockchain environments.

- Argument

Also referred to as soundness, this property states that an honest prover generating a valid proof will always be able to convince a verifier of its correctness. In other words, proof verification never results in a false negative. Conversely, a malicious prover is not able to forge a proof, if the statement to be proven is invalid.

This property is weakened to computational soundness for practical SNARK implementations, meaning that the likelihood for false negatives is negligibly small. The same applies to the forging of invalid proofs.

- Knowledge

  SNARKs extend the above definition of soundness to also include knowledge of a particular fact (*witness*) in the statement. A successful proof therefore not only guarantees that a computation was executed correctly, but also that the prover is in possession of such a witness. The properties required by the witness are specified in the computation itself.

In the same article, the authors modify their existing SNARK definition to also be **Z**ero-**K**nowledge, leading to the final zkSNARK construction discussed in this work. They do this by applying techniques used in existing non-interactive zero-knowledge proofs to hide the value of the prover's witness. Consequently, a verifier is only able to certify the inclusion of the witness in the requested computation, but remains incapable of its reconstruction.

Technical Background zkSNARKs can be realized on the basis of NP-complete decision problems. A decision problem in this context is any algorithm which, given a specific input, verifies a fact about that input by either answering "true" or "false". The complexity class NP (nondeterministicly polynomial) is defined as the subset of decision problems with efficiently verifiable witnesses.

A (efficiently verifiable) witness is any piece of information, which allows the decision algorithm to output "true" in polynomial time with regards to the input size. In the simplest case, the fact which needs to be verified by the decision problem can be modeled as an equation. Then any solution to this equation fulfills the definition of a witness.

To illustrate this, let us consider the *Boolean satisfiability problem (SAT)*, one of the most famous NP problems in literature: given a Boolean formula with operators $\wedge$ (and), $\vee$ (or), $\neg$ (not), e.g. $p \neg q \wedge r$, is there a variable assignment $\sigma : x \mapsto \{true, false\}$ which makes the formula itself evaluate to *true*?

In the general case, SAT is conjectured to not be solvable in polynomial time, especially when confronted with a sufficiently large formula. When offered a correct assignment, however, this correctness can easily be verified by simply plugging in the values for the variables and evaluating the resulting formula. Any correct variable assignment is therefore a witness for the decision problem. Note, that this definition of a witness is equivalent to the kind of witness used during a zkSNARK evaluation.

As mentioned above, deriving a SAT solution is thought to require a superpolynomial algorithm. This is, because SAT is also an NP-complete decision problem, meaning it is as hard as the hardest problem in NP. Whether or not there exist polynomial time algo-

rithms for NP-complete problems is one of the largest open questions in mathematics[1]. Nonetheless, most researchers believe that this is not the case.

This circumstance is exploited by the zkSNARK construction. A zkSNARK scheme takes a program as input and transforms the program into an NP-complete decision problem. It is important that this transformation is witness-preserving. As such, any witness (solution) to the original program also becomes a witness for the new NP-complete problem.

Due to the lack of polynomial time algorithms it is nearly impossible to find a witness for the transformed problem through mere brute-force calculation. By executing the original problem, however, such a witness can be constructed efficiently. A verifier can then check that the solution is indeed a witness for the given NP statement, without having to execute the program itself.

Solution Approaches One of the earliest practical implementation of a zkSNARK goes back to [44]. It is based on Quadratic Arithmetic Programs (QAP), an NP-complete problem involving solutions for quadratic polynomial equations (conf. 7.2). In their work, the authors formulate an algorithm for witness-preserving conversion of arithmetic circuits into QAPs. More precisely, the original witness, which is a correct variable assignment for the circuit, is transformed into solution vectors for the resulting QAP. Because the outputs of the circuit are also part of the variable assignment, such a witness essentially represents the computation modelled by the circuit. Therefore, by verifying that the transformed witness is indeed a solution to the QAP, any third party can be assured that the original witness must be a valid solution for the computation.

This approach is picked up by [17]. Besides making some adjustments concerning the underlying QAP construction, the authors make an important contribution towards general usability by introducing the first universal circuit compiler for zkSNARKs. Their compiler accepts any program written for traditional von-Neumann-architectures (with reduced instruction sets) and transforms them into arithmetic circuits. The compiled circuits are compatible with the previously described QAP transformation. As a result, zkSNARKs can be generated for arbitrary computer programs.

Another important landmark contribution is presented in [50]. The author introduces a novel approach for calculating pairing functions, a vital component needed when formulating QAPs. This new method, which is commonly referred to as *Groth16*, leads to significant improvements in performance for QAP-based zkSNARKs. It fact, these performance gains are so substantial, that Groth16 remains the most widely used zkSNARK in practice (conf. 8).

All QAP-based zkSNARKs are what is known as *preprocessing zkSNARKs*. This means, they require a setup process to generate auxiliary values for the QAP construction. Because these auxiliary values can also be used to forge proofs for invalid results, the setup must be carried out by a third party, which can be trusted to destroy the values after the setup has been completed. Assuming such a strong notion of trust greatly lowers the security guarantees made by the zkSNARK protocol.

Naturally, this behavior is undesirable, as it directly opposes our original motivation of creating trust in a trustless environment. Many newer schemes are therefore designed to be *transparent zkSNARKs*, meaning they do not require a trusted setup. Among others,

[1] http://www.claymath.org/millennium-problems

these include *Ligero* [5], *Hyrax* [94], *zkSTARKs* [15] and *Spartans* [88]. What all of these more recent solutions have in common, however, is greatly reduced performance when compared to the original QAP approach. This is especially true for proof size, which can be in the hundreds of kilobytes for even small programs. While this makes them currently impractical for blockchain applications, research is ongoing and transparent zkSNARKs may soon replace QAP-based ones as the status-quo.

7.2. Quadratic Arithmetic Programs

QAPs are mentioned for the first time in [44]. In fact, they are only a high-performance extension of *Quadratic Span Programs (QSP)* introduced in the same article with QSPs themselves being a special case of *Span Programs (SP)* [62]. We will give a brief mathematical definition for QAPs and explain how they can be used in an efficient zkSNARK scheme. For specific implementation details, the reader is referred to the original work.

7.2.1. Definition

QAPs offer an NP-complete language that allows anyone to check the correctness of a variable assignment for a given arithmetic circuit. As mentioned before, transformation exists to represent any fixed-size algorithm as an arithmetic circuit. As such, they serve as the foundation for many current zkSNARK implementations (conf. 8).

At their core, QAPs are built around quadratic equations of polynomials and various linear combinations thereof. More specifically, given the polynomials $\bigcup_{k=0}^{m} v_k(x)$, $\bigcup_{k=0}^{m} w_k(x)$, $\bigcup_{k=0}^{m} y_k(x)$ and a target polynomial $t(x)$, these fulfill the definition of a QAP for the arithmetic circuit f in the following case: There exist $a_1, \ldots, a_n, \ldots, a_{n'}, \ldots, a_m$, with

$$t(x) \text{ divides } \left(v_0(x) + \sum_{k=1}^{m} a_k v_k(x)\right)\left(w_0(x) + \sum_{k=1}^{m} a_k w_k(x)\right) - \left(y_0(x) + \sum_{k=1}^{m} a_k y_k(x)\right)$$

if and only if the input variables $a_1, \ldots, a_n$ and output variables $a_{n'}, \ldots, a_m$ are a valid assignment for f.

7.2.2. zkSNARK scheme

When using such a QAP for verifiable computing, it is the provers job to find the factors $a_1, \ldots, a_m$ that fulfill the above stated definition. Given such factors, it is easy to check that the polynomials are indeed a linear combination of $t(x)$. Speaking in terms of complexity theory, this is an NP-complete problem where the factors $a_1, \ldots, a_m$ act as the witness. On the flip side, this also means that likely no efficient polynomial time algorithm exists for finding appropriate indices, especially if the chosen polynomials become large enough.

The authors of [44] make this possible in the context of verifiable computation by constructing an algorithm that allows witness-preserving transformation of any arithmetic circuit into a corresponding QAP. Consequently, a prover may simply execute the arithmetic circuit f on the input values $a_1, \ldots, a_n$ to obtain the outputs $a_{n'}, \ldots, a_m$ and any intermediate values $a_n, \ldots, a_{n'}$. By way of this construction, the computation of f actually

yields the witnesses required to verify the QAP and because we know that f represents a fixed-size algorithm, this can be done efficiently.

Interestingly, the main contribution of the QAP construction is not its zero-knowledge property but rather its succinctness when compared to other zkSNARKs. To achieve this, a verifier does not check the entire polynomial equation but instead chooses a random evaluation point before the QAP is even constructed. This point is included in the *Common Reference String (CRS)* which also contains other auxiliary values and serves as a ground truth for the protocol. The authors of the QAP construction show that this approach is only negligibly less secure but leads to a significant reduction in proof size.

Implementing the zero-knowledge property, on the other hand, is almost trivial. Because of certain characteristics of the underlying polynomial equations, they exhibit a special case of homomorphism. As a result, the witness must not be included directly in the proof but can be hidden by multiplication with a constant factor. This does not, however, affect the correctness of the verification algorithm. Thus, f can still be verified even though its secret inputs are never made public.

8. Implementations

In this chapter, we will describe several implementations of the zkSNARK protocol. Note, that we focus on QAP-based zkSNARKs exclusively, since this is the only scheme mature enough to offer a broad variety of implementations to choose from.

8.1. Overview

zkSNARK compilation happens in two stages. First, the program, which is to be verified, must be converted into an arithmetic circuit. This circuit can then be used to repeatedly derive proofs for each individual instantiation of the program. In the case of QAP-based zkSNARKs, this second step also involves a one time trusted setup which must be performed before a proof can be generated.

When discussing zkSNARK implementations, this distinction becomes important, since most libraries focus on only one of these stages. Therefore, to obtain a full zkSNARK proof, both, a circuit compiler and a proof generator are needed.

Interchange Formats Several interchange formats have been developed to ease the coupling process between circuit compilers and proof generators. These formats allow seamless plug and play functionality for different zkSNARKs libraries, by providing a well-known textual representation of the program's underlying circuit.

Rank-1-Constraint-Systems (R1CS) are the de-facto standard in this realm. They are a mathematical representation of a circuit's architecture, describing all of its properties, including input variables, output variables and logic gates. Other formats include *Prover Worksheet* and *andytoshi*.

Special algorithm's exist, which can transform R1CS circuits into QAPs very efficiently. All implementations discussed in this section support the use of R1CS.

Proof Generators One of the earliest projects implementing the zkSNARK scheme was Pinocchio [81]. Based on the theoretical work of [44], Pinocchio is both, a proof generator and a circuit compiler for programs written in C.

Pinocchio's authors have also laid important groundwork by formalizing the properties of non-interactive proof systems based on zkSNARKs. Its architecture was later adopted by many of the more recent implementations.

Following the success of Pinocchio, the first libraries which only focus on proof generation started appearing. The biggest players in this space are libsnark[1], snarkjs[2] and

[1] https://github.com/scipr-lab/libsnark
[2] https://github.com/iden3/snarkjs

bellman[3], written in C++, JavaScript and Rust, respectively. All three libraries are under active development as of the time of writing.

Even though they all support the QAP-based approach discussed in this work, their author's have recently started to implement newer, more academic zkSNARKs in hopes of achieving better performance.

Circuit Compilers With the advent of pure proving libraries in the back end, more and more circuit compilers have started appearing for the front end of zkSNARK development. They make several adjustments to the construction process to either reduce the size of the resulting circuit or increase its execution speed. Additionally, they equip users with different auxiliary tools that make the development of zkSNARK programs closer to the traditional experience of writing software.

ZoKrates [40] is a full-stack circuit compiler that focuses on usability over performance. It comes with a fully functioning tool chain that lets users write zkSNARK applications in Python and host them on Ethereum with only a few commands.

Circom[4], on the other hand, is a pure circuit compiler created by the makers of snarkjs. It only accepts programs written in its own custom language inspired by hardware description languages such as VHDL. While less users might be familiar with this approach of defining circuits directly, it gives them full control over the design of the final circuit. When employing the right techniques, this can lead to considerable performance gains over circuit compilers with high-level languages.

xjSnark [67] is another pure circuit compiler which seeks to bridge the gap between usability and performance. Accepting programs written in Java, its aggressive compiler makes several passes to replace non-optimal code with pre-defined template circuits which have been optimized for use in zkSNARKs. It also includes its own IDE with semantic syntax checking.

8.2. Deep-Dive: ZoKrates

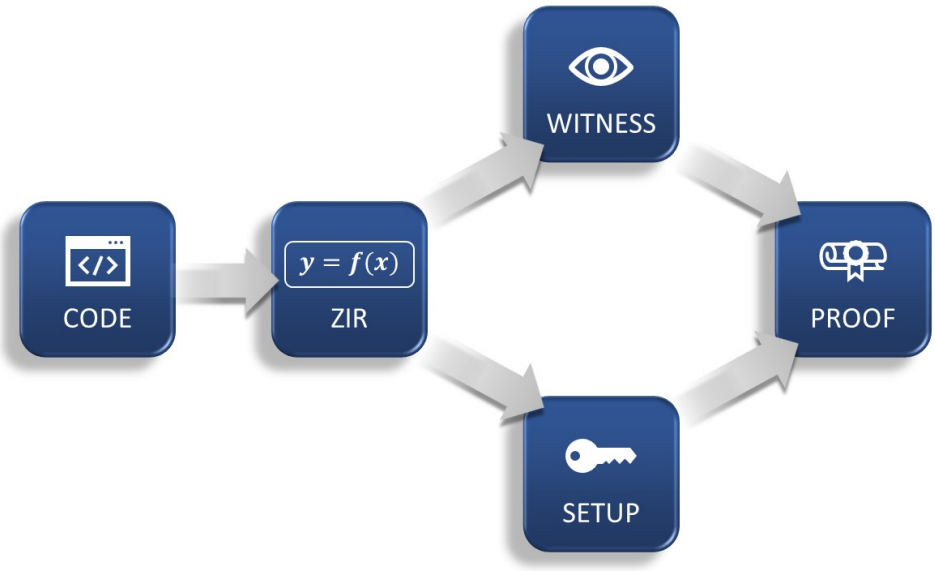
ZoKrates [40] is a comprehensive suite of tools for developing zkSNARKs on the Ethereum blockchain. It is designed to support the whole development process, starting with the initial program definition up to the final creation of the verification smart contract.

Programs are written in a dialect of the Python programming language. This dialect represent a heavily simplified subset of the original language which can be efficiently converted to arithmetic circuits. It further introduces new data types and other primitives specific to the zkSNARK setting.

ZoKrates uses a custom compiler to transform programs into their respective R1CS representations. The generation of the actual zkSNARK is done by libsnark according to the QAP approach outlined in [17]. After performing the trusted setup, ZoKrates uses the verification key to generate Solidity code for a smart contract which is able to verify proofs for this particular circuit. Provers can also use ZoKrates to execute the program and obtain its result along with a valid proof containing the witness.

[3] https://github.com/zkcrypto/bellman
[4] https://github.com/iden3/circom

Figure 8.1.: Flow diagram of the ZoKrates compilation process.



8.2.1. Architecture

ZoKrates is designed as a toolbox consisting of multiple standalone executables. All components are written in Rust. Their code is open-source and available on GitHub [5].

Compiler The first step for any zkSNARK is defining the computation which needs to be proven. ZoKrates developers complete this task by using the built-in DSL to create a custom program for their specific use case. Such a program is capable of supporting an arbitrary number of inputs and outputs. Furthermore, inputs might be "private", meaning they are hidden from verifiers with zero-knowledge techniques.

The ZoKrates compiler takes this program and converts it into *flattened code*. This internal representation is specific to ZoKrates. While similar to R1CS, it contains additional directives which still allow witnesses to be generated from just the flattened code alone, without need for the original program.

- Witness Generator

  Provers can use the witness generator to execute flattened program code. After supplying all of the program's public and private inputs, the witness generator computes the results and a witness for the specified values.

  The witness itself is a simple description of all the values which were assigned to either an input or an output variable during the computation. It is stored as a file for later use by the proving system.

- Setup and Proof Generator

[5] https://github.com/zokrates/zokrates

The main goal of ZoKrates is giving developers a platform where they can easily create zkSNARKs for their programs from start to finish. As such, ZoKrates only provides components which are not already available in other forms. In particular, it does not reimplement the generation of proofs from R1CS, as many libraries for this task already exist.

ZoKrates merely converts a program's flattened code into R1CS and passes it on to libsnark for building the final zkSNARK. After libsnark has executed the trusted setup, a prover can use the generated proving key to create proofs for any witness they have found. By reading the output file created by the witness generator, libsnark is thus able to create a zero-knowledge proof for this particular instantiation of the program. ZoKrates formats this proof in such a way that it can be directly checked by an on-chain verification contract.

- Contract Generator

  Because ZoKrates is designed for use in a blockchain environment, it also features a code generator for smart contracts. Using the verification key obtained from libsnark's trusted setup, the contract generator outputs a Solidity file containing an Ethereum contract with a single public function: *verifyTx*.

  Verifiers can send zero-knowledge proofs for a particular program directly to the verifyTx function of its dedicated verification contract and quickly obtain an answer without having to execute the verification themselves. The function accepts a computation's result, all of its public arguments and the proof supplied by the prover, and outputs either *true* or *false*, depending on the correctness of the computation.

  As a side-effect of using the blockchain as a publicly auditable verifier, verifyTx needs to be called only once. Since its result is permanently recorded, anyone can look at the function's output and see that the computation was executed correctly without having to re-executing the verification process.

- Circuit Importer

  ZoKrates' circuit importer acts as a small support tool for interoperability with other circuit generators. It is able to transform any circuit supplied as a R1CS back into ZoKrates-internal flattened code. This allows circuits generated by other means than the ZoKrates compiler (e.g. handcrafted) to be used by the toolchain.

  Note, that these flattened code objects, however, cannot be used for witness generation due to missing metadata which is normally added by the compiler (conf. 8.2.1).

8.2.2. Language

As mentioned before, the language used to write ZoKrates programs is a DSL based on Python syntax. Unlike Python source files, ZoKrates code may not contain any top-level instructions. Instead the code is structured as a collection of functions, with a traditional main function serving as the program's sole entry point. Functionality can be split across multiple source files. ZoKrates also comes with an extensive standard library covering common software patterns and the most important cryptographic algorithms.

Functions The main function accepts a variable number of arguments and can yield any number of return values. It is also the only function which may accept private arguments. Values for these arguments are hidden during compilation by leveraging the zero-knowledge property of the underlying zkSNARK. Verifiers may not extract them from the proof but can still observe that they were correctly supplied by the prover during proof generation.

An important distinction from pure Python code is the requirement of C-style forward declarations. This means that any function must be declared before it can be called. "Before" in this context refers to the function's lexical order within the source file. Once declared, functions can be called up to an arbitrary depth, however, recursion is not supported, since this would make the resulting circuit unbounded.

Variables Currently, the only supported data type for variables is an element of the zkSNARK's prime field. All other data types must be simulated by interpreting the variable, e.g. a Boolean variable could have it's value restricted to 1 and 0.

There also exists no support for global variables. All variables must be declared inside a function and are local to that function's scope alone.

Operations Just like Python, ZoKrates code defines operations imperatively as a series of instructions. Variables and constants can serve as operands for any of the four basic arithmetic operators $(+, -, *, /)$, or as inputs for binary comparisons $(==, <, <=, >, >=)$.

Comparisons can be used in the context of control flow structures or as assertions. Assertions require no special syntax beyond the comparison statement itself. A failing assertion will always lead to an invalid proof.

Control flow structures, on the other hand, are equivalent to what is found in most other languages. The compiler supports if-else-blocks and for-loops. Note, that all for-loops must be declared with an upper runtime bound, so that they can be unrolled during circuit generation. As is the case with recursion, circuit generation would be unable to complete otherwise due to the loop's unboundedness.

8.3. Deep-Dive: xjSnark

xjSnark [67] is a standalone circuit compiler for zkSNARKs. The code is open-source and available on GitHub[6].

It accepts programs written in a high-level language and converts them to R1CS. The language itself is a Java-based DSL implemented as a Java language extension with JetBrains MPS. It also features its own IDE, complete with syntax checker and code analyzer.

Since xjSnark merely builds the corresponding circuits for a given program, its compilation output must be processed further to generate an actual proof. This can be done by any zkSNARK library supporting the R1CS format, e.g. libsnark or snarkjs.

xjSnark differentiates itself from other circuit compilers by placing heavy emphasis on low-level optimization techniques specific to the zkSNARK setting. To this end, circuits

[6] https://github.com/akosba/xjsnark

are built incrementally by the compiler's multiple passes. After a skeleton circuit is constructed during its initial run, later passes fill in the code's functionality as pre-built language blocks.

These code blocks are highly optimized and often represent the state-of-the-art solution for a specific problem (according to the developers). This allows xjSnark to offer many advanced operations, which must typically be handcrafted in other zkSNARK DSLs, as built-in language constructs. Such operations include manipulation of individual bits, arithmetic for long integers (i.e. larger than the underlying prime field) and dynamic memory access.

As circuits can grow quite large as a result of these optimizations, xjSnark also applies several minimization techniques. This brings the resulting circuits back down to manageable sizes.

8.3.1. Language

An xjSnark source file consists of a single top-level *Program* declaration which is similar in nature to a Java class. All code must reside within the boundaries of this declaration.

Control-Flow Execution begins at a single mandatory main function. This function has no parameters or return values. Instead, inputs and outputs are defined via special statements at the top of the source file.

After invoking the main function, arbitrary functions in the same Program declaration can be called. Additional control-flow can be structured using if-else statements and bounded for-loops.

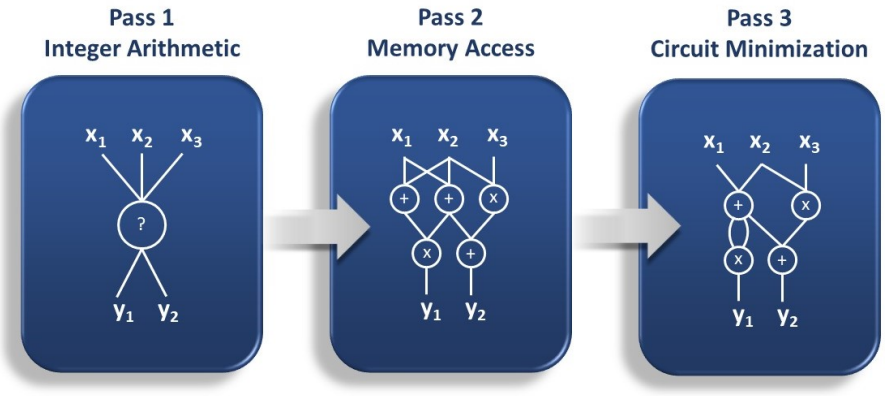
As an important new addition, xjSnark also introduces *external code blocks*. These blocks allow developers to call external Java functions from within the DSL. Since these external functions may contain arbitrary code, naturally, they cannot be included in the outcome of the circuit construction. Rather, they are a tool for developers to outsource the potentially expensive witness computation task to external libraries. This way, it can be included in the same source file as the proof circuit instead of another project as is the case with other circuit compilers.

Variables Variables in xjSnark exist on both, a global and a local level. Global variables are defined throughout the entire Program declaration, whereas local variables are only valid within the function scope where they were defined.

There are two data types available for new variables, both of which offer type-specific operations and have different overflow behavior. xjSnark allows both types to be used in the context of indexable arrays.

- Field elements are integers which overflow at a certain modulus. This modulus can be chosen by the programmer individually for each variable. They are efficiently implemented using the mathematical properties of the underlying zkSNARK's prime field.
- Unsigned integers are more akin to the integers found in traditional programming languages. They have a certain number of bits and overflow once all bits are ex-

Figure 8.2.: Functional overview of the xjSnark compiler passes.



hausted. Again, the number of bits can be specified by the programmer. It is worth pointing out, that these unsigned integers can also be thought of as a specialized case of field elements where the modulus is a power of two.

Operations As mentioned before, xjSnark natively offers many operations which are not commonly found elsewhere. Programmers can choose from the following:

- the usual arithmetic operations used for addition, subtraction, multiplication and division $(+, -, *, /)$,
- comparison operators to be used for conditions $(==, <, <=, \ldots)$,
- logical operators to concatenate conditions $(AND, OR, XOR, \ldots)$,
- field operations for field elements, such as finding the inverse of an element,
- bit-wise operations for unsigned integers $(AND, OR, XOR, \ldots)$.

8.3.2. Optimization Techniques

To implement its various optimization techniques, the xjSnark compiler must scan a source file multiple times. The output circuit is then generated incrementally during every iteration. Each pass gives the compiler a chance to analyze the code structure after the previous step and lets it improve further upon the constructed circuit.

First Pass: Integer Arithmetic During the first pass, only a high-level dummy circuit is built from the various arithmetic operations found in the source file. The main purpose of this pass is to limit range conversion of overflown variables.

Because range conversions are very expensive in general, as they involve a high number of multiplication gates in the circuit, the compiler tries to eliminate them as much as possible. Consequently, values are allowed to exceed their range in most cases and only converted back when this would affect the outcome of an operation.

An example of an unaffected operation would be integer addition. Adding two integers always produces a correct result, even if one or both are larger than the size limit declared by the developer. For multiplications, on the other hand, values must be brought back into range, since multiplicative properties do not hold when e.g. the factor exceeds its size limit.

When implementing overflow behavior, the compiler also distinguishes between short and long integer arithmetic. Any integer with a modulus smaller than the square root of the zkSNARK's field modulus is considered a short integer. Because short integers are so small, the result of any operation involving two short integers is guaranteed to fit onto a single wire in the circuit.

Integers with a larger modulus are considered long integers. Their representation in the circuit must be split across multiple wires, leading to a more involved overflow conversion. While this detail is hidden from the developer by the DSL, it still affects the performance of the final proof.

Second Pass: Memory Access After all arithmetic operations have been processed, the compiler initiates a second pass to generate the necessary circuitry for dynamic memory access. It analyzes usage behavior for different memory locations and chooses from four different implementation techniques to achieve the best possible performance.

In most cases a *naive linear scan* suffices, as long as the number of elements and random accesses remains relatively low. For higher numbers of random access operations, the authors found that a *custom permutation network* is more efficient when modeling the memory locations in the circuit. Should the number of elements exceed a certain threshold, however, the performance of the mentioned approaches is diminished severely. In this case, *Merkle tree proofs* become a possibility, since they allow comparatively inexpensive verification of access to a large field of memory locations.

Finally, the xjSnark developers also provide their own solution for memory that is read-only. Instead of checking such locations directly in the circuit, they are verified along with the zkSNARK proof. This is done by including a mapping of the memory locations in the witness for the computation. The resulting witness is only accepted if all locations have been accessed correctly. This approach greatly outperforms the other three in the case of read-only memory.

Third Pass: Circuit Minimization After completion of the second pass, the circuit is already in a usable state. An optional third pass may be added to shrink its size for better performance and an easier deployment. This is done by further reducing the number of multiplication gates needed for bitwise operations. The technique behind this is called multivariate polynomial minimization [56]. It works by exploiting certain characteristics that exist when variables of a bitwise operations are modelled as polynomials. The developers of xjSnark state that they focus on bitwise operations because many cryptographic primitives rely very heavily on them, and having efficient circuit implementations would enable important zkSNARK use cases.

Part IV.

Secure Multi-Party Computation

9. Concept

In an ideal world, we would like our computations to be a) correct, b) private, and c) include no single point of failure. *Multi-Party Computation (MPC)*[1] promises such an ideal world. The technology was first mentioned in the context of Yao's famous Millionaires' problem [98], where two millionaires envision a protocol to determine which is richer without divulging their real wealth.

In a more general sense, MPC refers to a wide variety of different security protocols with the same core principle: The participants provide secret inputs, which are hidden using secure obfuscation schemes, and jointly execute operations on these values without revealing them. At the end of the protocol, its final output is made public. By looking at certain cryptographic artifacts created during runtime, the participants can later verify that the computation was indeed executed correctly.

Throughout MPC's short history, most implementations have been realized using at least one of the following four concepts [105]:

- Garbled Circuits [98],
- Oblivious Transfer [86],
- Linear Secret-Sharing [89],
- Fully Homomorphic Encryption [46].

Of the listed technologies, linear secret-sharing finds the most use in real-world applications. Yao's original garbled circuits technique was only designed to serve two parties but it has since been extended to the n-party case. Nonetheless, garbled circuits exhibit poor performance at scale [79], making them inferior to other MPC schemes. Oblivious transfer, on the other hand, is rarely used by itself and can instead be found in many existing protocols [31, 10, 90], as a way to secure communication channels.

Lastly, fully homomorphic encryption (FHE) is an ongoing research effort that has gained much traction in recent years. In theory, FHE enables arbitrary computations on encrypted user data, preserving the applied operations when the data is finally decrypted. While this already sees some use within certain limits, performance remains poor for now as well [35, 106].

[1] sometimes called *secure Multi-Party Computation (sMPC)*

10. Background

We proceed by describing two popular versions of linear secret-sharing. Both find use in the implementations we discussed in the next chapter.

10.1. Shamir's Secret Sharing

Shamir [89] presents one of the earliest linear secret-sharing algorithms (SSS), which is still in use today. The scheme, which was originally intended for secure storage of cryptographic keys, is also suitable for MPC because of its homomorphic properties. Specifically, the chosen representation of the secret-shared information exhibits an additive and multiplicative homomorphism [106].

Theory The author describes SSS as an *(k, n) threshold scheme*, meaning that the data D is split into n pieces. Reconstruction of D requires at least k out of the original n data pieces. Possession of anything less than k pieces, however, reveals absolutely no information about D. In the case of SSS, Shamir assumes an honest majority and consequently sets $n = 2k - 1$. This way, an attacker always holds less shares than the minimum number required to reconstruct D.

The sharing algorithm is founded on the principles of polynomial interpolation. It assumes that D can be treated as a number, e.g. via binary representation. For practicality reasons, the algorithm also operates on a prime field $\mathbb{F}_p$, where p is a prime $> D$ and $> n$, instead of all natural numbers $\mathbb{N}$.

The central idea is, that for a given series of points $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$, there exists exactly one polynomial $q(x)$ of degree $k - 1$, s.t. $q(x_i) = y_i$. One such polynomial $q(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{k-1} x^{k-1}$, acts as the encryption key. It is picked at random during a trusted setup phase (e.g. by a trusted third party), under the restriction that $a_0 = D$. The individual data pieces then are defined as $D_i = q(i), \forall i \in [n]$.

Reconstruction Given at least k pieces, reconstruction of $q(x)$ is possible with polynomial interpolation, for which many algorithms exist. After the participants have reconstructed the polynomial, they may trivially obtain the shared value D by evaluating $q(0)$.

As an important detail of polynomial interpolation, no information can be obtained with anything less than k shares of D. When operating on $\mathbb{F}_p$, there are p possible values for D. An attacker in possession of $k - 1$ pieces, can construct exactly one polynomial $q'(x)$ for each of these values, s.t. $q'(i) = D_i$. Due to the construction process, all p polynomials are equally likely. Hence, the attacker learns nothing unless they can obtain another share of D.

SSS features several advantages over previous secret-sharing schemes. Most notably, new pieces can be dynamically added by evaluating $q(x)$ at the given location. Conversely,

by changing the underlying polynomial $q(x)$ and redistributing its shares, older pieces may be invalidated.

The scheme further allows for hierarchical key distribution. As a participant obtains more shares of D, their relative voting power grows. A small clique of parties with many shares thus outweighs a majority with fewer shares. This property is useful when modeling power structures with different security levels, e.q. internal company networks.

10.2. SPDZ

SPDZ (pronounced *speedz*) [35] is a relatively recent multi-party protocol for secure general-purpose computations based on additive secret sharing. It supports arbitrary addition and multiplication operations over a field $\mathbb{F}_{p^k}$, where p is prime and k an integer.

The protocol is capable of handling inputs from multiple joint parties, but always yields a single aggregated computation result. Authenticity of the inputs and any intermediate values obtained during execution is guaranteed by an embedded cryptographic MAC. The developers of [35] claim that this MAC makes the protocol secure in a scenario with n passive adversaries, or even $n - 1$ actively malicious adversaries, where n is the total number of participants in a round.

SPDZ promises significant performance increases over earlier MPC implementations. Its runtime complexity is projected to be independent of the computation inputs or even the evaluated function itself. The significant increase in performance is accomplished by splitting the algorithm into two distinct phases. Function execution is preceded by an offline preprocessing phase which can happen well in advance of the actual computation. During this offline phase, the participants generate a set of auxiliary values via *Somewhat-Homomorphic-Encryption (SHE)*. These values are later used at various points during the online computation phase to speed up its execution.

SHE is computationally expensive but the values only have to be generated once and can be reused for multiple rounds. By front-loading expensive work to a preprocessing phase, SPDZ is thus able to lower the complexity of its online phase. Whereas earlier MPC implementations ran in quadratic computation- and communication complexity with respect to the number of participants, SPDZ reduces this to linear.

10.2.1. Theory

Participants of the SPDZ protocol need to interact with a variety of numeric values, e.g inputs, intermediate results, MACs, etc. For confidentiality purposes, all of these values are represented using a linear secret-sharing scheme during computation. Before their contents can be read in the clear, these representations must be opened and their MACs verified. SPDZ distinguishes between two different value representations with distinct verification mechanisms.

Publicly Verifiable Secrets Most secret values in SPDZ are publicly verifiable. This includes computation inputs, intermediate results and also some of the auxiliary values generated during preprocessing. The representation of such values contains a MAC that is generated with a unique global key. When opened, the contained value is reconstructed

concurrently by all nodes. After obtaining the secret value, participants in possession of the global MAC key can check whether it was manipulated during any of the previous computation steps.

More specifically, a value a has a publicly verifiable representation triple

$$\langle a \rangle \coloneqq ((a_1, a_2, \ldots, a_n), (\gamma(a)_1, \gamma(a)_2, \ldots, \gamma(a)_n), \delta),$$

where a_n is the secret value share and $\gamma(a)_n$ the secret MAC share held by the n-th party. Because SPDZ uses simple additive secret-sharing,

$$a = \sum_{i=1}^{n} a_n, \qquad \text{and} \qquad \gamma(a) = \sum_{i=1}^{n} \gamma(a)_n.$$

This form of additive secret-sharing is linear. It allows for addition and multiplication operations on the shares without violating the rules for reassembly. The public modifier δ is necessary during certain operations involving constant values to preserve linearity of the MAC. More details on this will be given in 10.2.1.

To open a publicly verifiable value, the shares a_n must be exchanged by all of the protocol's participants. Once every party has received their full set of shares, they can reconstruct the original value by summation as stated above. The embedded MAC can be used to verify the obtained value and any of the computation steps during which it was involved. If α is the global MAC key, then the MAC is defined as $\gamma(a) = \alpha(a + \delta)$. When a value representation is first created, $\gamma(a)$ is set equal to this MAC, with a referring to its original value and $\delta = 0$. $\gamma(a)$ is then secret-shared among the participants. Since $\gamma(a)$'s representation is linear, the MAC relationship continues to hold, even when the underlying value a is subject to repeated mathematical operations.

The verification process starts, after all shares $\gamma(a)_n$ have been exchanged and $\gamma(a)$ reassembled by the participants. They can then verify whether a was manipulated by checking if

$$\sum_{i=1}^{n} \gamma(a)_n = \gamma(a) = \alpha(a + \delta).$$

The probability, that this equality still holds in the case where a malicious party has manipulated their share of a or skipped an execution step is negligible. Therefore, if the verification succeeds, the representation was opened correctly and its value can be trusted.

Privately Verifiable Secrets A handful of auxiliary values encountered during a given SPDZ run can be opened and checked by a single participant using only a personal key. Such values are said to be privately verifiable. The main purpose of private verification is opening a secret-shared version of the global MAC key α. Private verification is also possible for some other values obtained during preprocessing, many of which are used as nonces which randomize certain parts of the protocol. Most importantly, any privately verifiable value can also be verified publicly by simply repeating the opening process for every participant.

Given a value a, its privately verifiable representation is

$$[\![a]\!] = ((a_1, a_2, \ldots, a_n), (\gamma(a)_1^i, \gamma(a)_2^i, \ldots, \gamma(a)_n^i, \beta_i)_{i=1,2,\ldots,n}).$$

Again, a_n is the secret value share in possession by the n-th party, with

$$a = \sum_{i=1}^{n} a_i.$$

This time, however, there are n different MACs, one for each of the n participants. There is also no public modifier δ. This means that linear operations are not possible for $[\![\cdot]\!]$-shared values. For this reason, SPDZ only employs this representation for certain auxiliary values where the main objective is unconditional secrecy and no further MPC computations are required.

A private MAC representation consists of the personal key β_i and values $(\gamma(a)_1^i, \ldots, \gamma(a)_n^i)$, with

$$\gamma(a)^i = \sum_{j=1}^{n} \gamma(a)_j^i.$$

As a critical detail, each share $\gamma(a)_j^i$ belonging to the i-th MAC is held by the j-th party. Private verification therefore still requires all other parties to supply their MAC shares to the opening node. Forcing all nodes to cooperate during a private opening ensures that all verification activities are still authorized by the network.

The MAC itself is defined as $a\beta_i$ for participant i. At the beginning of the protocol, SPDZ sets $\gamma(a)^i = a\beta_i$ and derives the MAC shares from this. Therefore, when participant i privately opens a secret value, they can check its integrity by verifying that

$$\gamma(a)^i = \sum_{j=1}^{n} \gamma(a)_j^i = a\beta_i$$

still holds. For participants to be able to trust this MAC scheme, however, SPDZ must guarantee that all values were generated correctly. Therefore, the MAC shares as well as the private keys are created during the protocol's preprocessing stage subject to strict cryptographic security restrictions. It follows that any value using the $[\![\cdot]\!]$ representation must already be available at the beginning of the protocol. This is the reason why this representation cannot be chosen for anything other than rudimentary helper variables.

Computational Model SPDZ converts arithmetic circuits to a sequence of linear operations which can be solved using MPC. The underlying secret-sharing scheme supports the following operations:

- addition of two secret values $\langle a \rangle$ and $\langle b \rangle$
- multiplication of two secret values $\langle a \rangle$ and $\langle b \rangle$
- addition of a public constant e and a secret value $\langle a \rangle$
- multiplication of a public constant e and a secret value $\langle a \rangle$

According to [14] these operations are sufficient to model any arbitrary computation as an arithmetic circuit.

Addition and multiplication with a constant are defined as component-wise operations on the $\langle\cdot\rangle$-representation of a secret-shared SPDZ value. Therefore, trivially

$$\langle a\rangle+\langle b\rangle = ((a_1+b_1, a_2+b_2, \ldots, a_n+b_n), (\gamma(a)_1+\gamma(b)_1, \gamma(a)_2+\gamma(b)_2, \ldots, \gamma(a)_n+\gamma(b)_n), \delta_a+\delta_b),$$

and

$$e * \langle a\rangle = ((e * a_1, e * a_2, \ldots, e * a_n), (e * \gamma(a)_1, e * \gamma(a)_2, \ldots, e * \gamma(a)_n), e * \delta).$$

Adding a public constant to a shared value requires use of the public modifier δ. δ acts as a sort of error-term for constant additions to keep the representation linear. Its value is initialized to 0 and tracked publicly by all nodes. Addition can now be defined as

$$e + \langle a\rangle = ((e + a_1, a_2, \ldots, a_n), (\gamma(a)_1, \gamma(a)_2, \ldots, \gamma(a)_n), \delta - e).$$

Note, that the MAC stays unaltered. During verification, the verifier opens and checks a against its MAC. They then apply the difference accumulated by δ to obtain the real value of a. Since δ is being tracked publicly by all nodes, it cannot be manipulated without them noticing. An attacker can also infer no information about a from this public variable, as it merely represents a deviation from the true result.

The last remaining operation, multiplication of two secret values, is much more involved than its counterparts. Because multiplication cannot occur directly without revealing the secret value, SPDZ utilizes a number of dummy values with known multiples and solves the computations through addition. These dummy values come in triplets of the form $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ with $\langle c\rangle = \langle a\rangle * \langle b\rangle$. Just like the MACs, these multiplication triplets are generated during preprocessing under secure conditions. Because the required number of triplets cannot be known prior to the computation, a sufficiently large amount must be generated.

Multiplication of two secret values $\langle x\rangle, \langle y\rangle$ proceeds as follows:

$$\begin{aligned}\pi &= \langle x\rangle - \langle a\rangle,\\ \phi &= \langle y\rangle - \langle b\rangle,\\ \langle x\rangle * \langle y\rangle = \langle z\rangle &= \langle c\rangle + \pi\langle b\rangle + \phi\langle a\rangle + \pi\phi.\end{aligned}$$

Note, that π and ϕ are opened by the participants and can thus be represented by public constants. As long as the multiplicative property of the dummy triple holds, $\langle z\rangle$ can therefore be obtained with only the operations already established. Further, by not opening $\langle x\rangle$ or $\langle y\rangle$ directly, an attacker learns nothing about their real values.

Due to technical limitations of the preprocessing phase, a malicious node may introduce small errors during generation of the multiplication triplets. More specifically, an erroneous triplet violates the requirement that $\langle c\rangle = \langle a\rangle * \langle b\rangle$. Consequently, every triplet must be verified before use. This is done by sampling a throw-away triplet $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ with $\langle h\rangle = \langle f\rangle * \langle g\rangle$ and comparing the difference of their individual components. Given a random value $[\![t]\!]$ obtained from preprocessing, the participants compute

$$\begin{aligned}\sigma &= \langle b\rangle - \langle g\rangle,\\ \rho &= t\langle a\rangle - \langle f\rangle,\\ \Delta &= t\langle c\rangle - \langle h\rangle - \sigma\langle f\rangle - \rho\langle g\rangle - \sigma\rho.\end{aligned}$$

Intuitively, as long as the component-wise difference Δ between the triplets is 0, their errors are equal. To subvert the protocol, an attacker would therefore have to introduce the same-size error in both triplets. Since they are chosen independently at random, however, this event is probabilistically negligible.

10.2.2. Operation

As mentioned above, SPDZ operates in two phases. An expensive preprocessing phase is used to generate auxiliary values. Participants execute this first phase offline and asynchronously. The actual computation occurs synchronously during the online phase. Due to the pre-computed auxiliary values, this second step is fast and cheap. Because of the amount of technical background knowledge involved, we reverse the order of these phases in our explanation.

Online Phase To start a round of SPDZ, each participant first needs to transform their inputs into secret-shared values. To this end, preprocessing supplies them with a number of randomization variables r available as both, $\langle r \rangle$ and $[\![r]\!]$. A participant wishing to partake in the protocol first opens $[\![r]\!]$ in private and calculates the difference ϵ between r and their input. After publishing ϵ, the other parties can compute $\langle r \rangle + \epsilon$ to obtain a secret-shared version of the original input without knowing its value.

After all inputs have been shared, parties can start to execute a function via the described operations. MAC verification is postponed until the final result is obtained. Even though this behavior might lead to wasted computations, it is nonetheless important, since verifying the MAC requires knowledge of its global key α. Once α is known, anyone can forge a secret-shared value for the current round.

To prevent this, participants engage in what is known as a *partial opening* every time a secret-shared value must be opened. In this simplified opening process, one node is chosen as the leader, receives all shares and relays the reconstructed variable. Whether the broadcast value is legitimate, is only established during the final output phase.

After the function result has been computed, a commit-and-reveal scheme is used to verify all opened values simultaneously. First, the participants construct a randomized linear combination of all opened values with parameters obtained from preprocessing. Each participant also applies the same linear combination to their MAC shares of the opened values. They then irrevocably commit the combined values and MAC shares in a secure way[1]. $[\![\alpha]\!]$ is publicly opened. Because of the MAC representation's linear properties, the same check described in 10.2.1 can be applied to linear combinations of secret-shared values. Given the opened values $a_{1..m}$ and the random value e, verification succeeds if

$$\sum_{i=1}^{n}\sum_{j=1}^{m} e^j \gamma(a_j)_i = \alpha(\sum_{j=1}^{m} e^j(a_j + \delta_j)).$$

Offline Phase Prior to executing the computation, SPDZ participants engage in a labor-intensive preprocessing phase. By employing a combination of SHE and zero-knowledge-

[1] The commit-and-reveal scheme is not specified in [35], but many possible implementations exist. [13] suggests Pedersen-commitments.

proofs, they generate secret-shared parameters necessary for the online phase. The parameters are

- the global MAC key $[\![\alpha]\!]$,
- a randomization pair $(\langle r \rangle, [\![r]\!])$ for every input,
- a sufficiently large amount of multiplication triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$,
- a random value $[\![t]\!]$ to verify multiplication triplets,
- a random value $[\![e]\!]$ to form a linear combination of the opened values during the final output phase.

Note, that none of the generated parameters are tied to the exact values of the inputs or the executed function. Therefore, participants can conduct the preprocessing well in advance of the actual computation, even before its inputs are known. This two-phase split is what gives SPDZ its significant speed advantage over previous MPC implementations. Additionally, large parts of the offline phase can be parallelized, leading to further performance gains.

During the offline phase itself, participants rely on a homomorphic encryption scheme to generate random values from multiple inputs. Authenticity of the inputs is secured using a zero-knowledge scheme. Implementation details of the preprocessing phase are well beyond the scope of this work. The interested reader is referred to [35], sections 3-6 for a description by the original authors.

10.2.3. Public Auditability

One drawback of SPDZ is the fact, that its correctness can only be checked by active participants. Parties which do not directly contribute during a protocol run, have no way to verify the computation.

[13] presents an extension to classical SPDZ that solves this problem. By making a few adjustments, the authors transform SPDZ's architecture from a network of compute nodes to a client-server model. While the virtual "server" is still formed by the compute nodes, they now accept inputs from other nodes, which do not actively participate in the protocol (clients). Additionally, they also broadcast publicly auditable correctness proofs for each computation step. A public bulletin board permanently stores these proofs so that they can be used by the clients or anyone else to verify the computation even after its execution.

The proofs have their mathematical foundation in *Pedersen-commitments* [82]. A Pedersen-commitment of value a with randomness $\tilde{a}$ is defined as

$$pc(a, \tilde{a}) = g^a h^{\tilde{a},}$$

where g and h are global constants derived from a common reference string. Unlike traditional commitment schemes, e.g. hash functions, Pedersen-commitments are linear. Specifically, when a value is manipulated by linear operations, its commitment can be obtained by applying the same operations to the commitment of the original value. This property makes Pedersen-commitments an excellent fit for MPC.

To make use of this new verification scheme, a Pedersen-commitment is added to the representation of $\langle\cdot\rangle$-secret-shared values:

$$\langle a\rangle_A \coloneqq (\langle a\rangle, \langle\tilde{a}\rangle, pc(a,\tilde{a})).$$

As such, the commitment serves the same role as the embedded MAC, only from the standpoint of public auditors. Operations are redefined as follows:

$$\begin{aligned}\langle a\rangle_A + \langle b\rangle_A &= (\langle a\rangle + \langle b\rangle, \langle\tilde{a}\rangle + \langle\tilde{b}\rangle, pc(a,\tilde{a}) \cdot pc(b,\tilde{b})),\\ e + \langle a\rangle_A &= (e + \langle a\rangle, \langle\tilde{a}\rangle, pc(e,0) \cdot pc(a,\tilde{a})),\\ e \cdot \langle a\rangle_A &= (e \cdot \langle a\rangle, e \cdot \langle\tilde{a}\rangle, pc(a,\tilde{a})^e).\end{aligned}$$

Note, that the operations on the Pedersen-commitments are not applied directly during the computation phase. Instead, a client wishing to verify an opened result $(y, \tilde{y})$ locally applies all operations to the initial commitments stored on the public bulletin and checks whether the calculated commitment is equal to $g^y h^{\tilde{y}}$.

The initial commitments are produced during preprocessing. Each of the randomization inputs $\langle r\rangle$ from the original online phase is generated as $\langle r\rangle_A$, where $\tilde{r}$ is also randomized. All instances of $pc(r,\tilde{r})$ are published to the bulletin where they serve as a starting point for public audits. From there on, any new commitments are only generated through one of the listed operations. Multiplication of two secret values is derived from these operations in the same way as in classical SPDZ. The intermediate values π and ϕ obtained from the multiplication triplets are also stored on the bulletin, since clients need them to trace the execution path.

11. Implementations

We proceed by giving a short overview of current MPC solutions designed to work with blockchain environments. For each example, we list the employed MPC primitives and explain what differentiates it from the others.

11.1. Overview

As mentioned before, most real-world MPC systems rely on linear secret-sharing as their main cryptographic protocol. The most prominent example of this is the *Enigma* platform [106, 107]. Enigma leverages the properties of Shamir's secret-Sharing and SPDZ to build a privacy-preserving execution engine for smart contracts. In combination with a classical blockchain, this gives users the ability to have contracts with both, private and public portions.

The authors of [79] introduce *Pandora* as an extension to Enigma. They adopt large portions of the computation engine, but make several contributions to the underlying network code. This further increases the performance of outsourced computations.

Another system using linear secret-sharing but with a different methodology is *Keep* [73]. It provides a distributed market for verifiable computations where buyers and sellers of computational power come together. Designed from the ground up as a black box service, Keep's main goal is ease of users for its clients. As such, clients may purchase execution time on worker nodes as abstract execution containers (the eponymous "keeps"). When a purchase occurs, a new keep is spawned for this specific instance of the computation. An underlying SPDZ protocol ensures that the computation is executed securely and privately.

As for implementations of other MPC schemes, options are more limited. The authors of [83] suggest an unnamed system based on oblivious transfer and homomorphic encryption, but their work is still academic and very high-level. A more practical alternative on the basis of oblivious transfer and garbled circuits is introduced in [18]. By limiting their application to permissioned blockchains, the authors are able to circumvent many problems encountered by other MPC systems. Due to the better performance of permissioned environments, necessary data can be stored directly on-chain. Similarly, computations can be run as part of the built-in endorsement phase which replaces the consensus algorithm of permissionless networks. However, since we only consider verifiable computation techniques applicable to public blockchains, we will not pursue this approach further.

11.2. Deep-Dive: Enigma

Enigma [106, 107] is the name of a verifiable computation solution for blockchains developed at MIT. Its creators promise scalable and privacy-preserving off-chaining of general-purpose computations by leveraging recent advancements in MPC. The system itself is

blockchain-agnostic. Any blockchain providing smart contract functionality can serve as the basis for an Enigma instance.

Computations in Enigma are formulated using private smart contracts written in a dedicated scripting language. An interpreter splits these contracts into public and private sections. The public parts are executed in the regular blockchain environment under the security assumptions of the utilized consensus protocol. Private sections, on the other hand, contain references to specialized Enigma functions. These functions are executed only by a small network of compute nodes. Their inputs and execution flows are kept private using various cryptographic primitives.

The platform is built on top of a hierarchical storage architecture. There are three separate storage layers that fulfill different purposes. Any small amount of public data can be stored using the pre-existing blockchain infrastructure. This includes public computations, execution proofs, access rights, etc. A distributed hash-table serves as a storage for large chunks of binary data. This data can be accessed from the blockchain through references without including it in a block. Lastly, an MPC protocol based on SSS and SPDZ provides a secure storage interface for small, private values. MPC techniques can be used to include these values in computations without revealing them.

Enigma further relies on a built-in incentive scheme to guarantee participation and fairness. Compute nodes are rewarded for their services with transaction fees directly collected from the clients. Security deposits serve as a form of DoS protection and ensure that compute nodes have no incentive to cheat.

11.2.1. Shared Identities

The authors of [106] make the distinction between public and private blockchains. In particular, they note how members of private blockchains can uniquely identify each other by slightly relaxing the trust guarantees of public blockchains. In an attempt to bridge this gap, Enigma introduces the concept of *shared identities*. A shared identity represents a group of nodes with a common interest in a specific data set. Nodes within an identity have different roles in relation to the data, granting them different sets of permission. These roles are not fixed and can be changed dynamically.

On a technical level, shared identities are realized as an access-control list. The public keys of the participant nodes are taken and stored under a specific contract address on the blockchain. Other private smart contracts now have the ability to refer to this shared identity when checking the access rights of an incoming transaction.

It is important to note, that shared identities hold no innate permission logic. They are simple collections of keys. Instead, the executing contract is responsible for providing the necessary checks. In Enigma, the identity's creator is considered the owner by default, possessing full read- and write-access to the data. The other members may only have read-permissions. Developers are free to introduce their own access-control model, however. This gives them fine-grained control over who can see and manipulate a contract's private data.

Because re-implementing the default permission logic for every new contract would be error-prone, Enigma provides a public smart contract for this task. Private contracts can call this function using only the relevant public keys and reliably receive an access-granted or access-denied message.

11.2.2. Storage Model

All private smart contracts in Enigma have access to three different data stores. These stores are not replicated across contracts, but exists as global singletons shared by the entirety of the network. Each store relies on a different technology and makes different trade-offs concerning performance and security. When creating a new contract, clients therefore have to choose wisely which store is best suited for each contained datum.

Enigma's built-in scripting language contains a well-defined API that gives access to these stores from within a contract. This API is modeled after the widespread dictionary paradigm known from common programming languages. The set-operation takes a new *(key, value)*-pair and inserts it into the store's internal memory. Stored values can later be retrieved with the get-operation by using the key that was originally used to insert them. Some stores also feature additional API calls which can be used to access implementation-specific functionality. These calls will be covered in their store's respective section.

Public Ledger Every Enigma instance has access to a public append-only ledger. This ledger is simply the underlying blockchain implementation. It serves as permanent storage for any public read-only data that is needed during computations. This includes shared identities, the public permission contract, any non-sensitive contract data, as well as references to data in other stores. These values are stored permanently. Once a datum has been published to the ledger, it cannot be changed or overwritten.

The ledger is also used to store the proofs of correctness generated by the secure MPC protocol. More specifically, it serves as the public bulletin board for the SPDZ implementation that is the foundation of Enigma's computation phase (conf. 11.2.3). This detail is hidden by the MPC interpreter. A contract developer therefore does not have to call the ledger's API directly to store the obtained SPDZ proofs. Instead, the ledger is accessed passively by the interpreter during an MPC invocation.

Distributed Hash-Table Enigma uses a distributed hash-table (DHT) based on the Kademlia architecture [74] to complement its public ledger. While also storing data in a distributed way, unlike blockchains, DHTs do not replicate these values across all nodes. Instead, the data is separated into smaller chunks, which are then shared among the participants. A client can now query the network for any missing chunks to reassemble the original data. By avoiding much of the redundancy present in blockchains, DHTs are thus able to decrease the total amount of needed storage.

DHTs also lack the slow and expensive consensus algorithm that blockchains run to commit new data. Once a value is inserted, there are no further integrity guarantees that protect it from manipulation by the storage nodes. Neither does a DHT guarantee the availability of a stored datum. If all nodes carrying a specific chunk go offline, the value is lost. Clients must therefore rely on the honesty of the network.

By forgoing the strict security measures of blockchains, DHTs achieve significantly better read- and write-speeds than any current blockchain implementation. Since entries do not have to be replicated, they are also capable of storing large amounts of data at a much lower computational cost. Clients and private smart contracts can use the DHT to externally store data sets that would otherwise be too large for the public ledger. Additionally,

they can make use of symmetric key encryption to protect private data before submission. This way large data sets can be off-chained even if they contain sensitive information.

Enigma's DHT also stores the value shares used during MPC. Transferring their shares to the DHT lets share owners delegate computations to the storage nodes. Furthermore, since each storage node only holds a discrete view of the DHT without access to its entire data set, the shares cannot be reassembled by them.

Data stored in the DHT can then be accessed from the ledger or during an MPC computations. Instead of the complete data set, however, only a reference to it is manipulated. This reference exists in the form of the API's storage key. Access is restricted by a control mechanism based on the shared identities introduced earlier.

When storing a value, its owner has the option to also supply a predicate to the DHT's set-function. This predicate is evaluated for every DHT transaction. Only if the predicate evaluates to "true" for the requesting node does the DHT relinquish its data. It is constructed using a built-in predicate language that operates on the public keys of shared identities. Recall that private smart contracts use a similar mechanism to verify a nodes identity (conf. 11.2.1).

Multi-Party Computation Enigma's main value proposition – performant and privacy-preserving off-chain computations – is largely achieved through SPDZ-based MPC. This technology relies on a number of relatively complex protocols, especially during the offline phase.

To ease the setup process for smart contract creators, the Enigma developers chose to hide the intricacies of the underlying MPC protocol behind a simple key-value-store interface. This interface features an API similar to the public ledger and DHT data stores. Using the set-function, clients can insert a value under a certain name, whereas the get-function expects a name and retrieves a reference to its corresponding value.

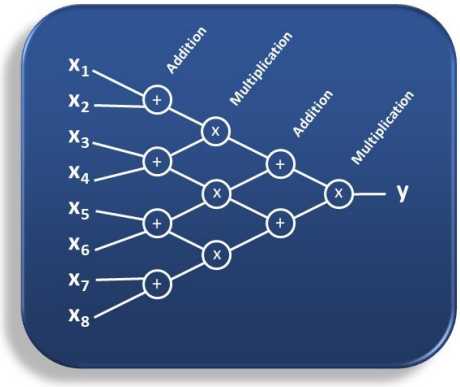
As one might expect, the advantage of using the MPC store over the other two stores, is the fact that one can include the contained data in computations without revealing it. Any datum inserted into the MPC store, is immediately secret-shared among the network. The participants for this are chosen at random.

In accordance with the security model of MPC, only its owner is able to reconstruct the original value. This is accomplished with the API's implementation-specific declassify function. Access for computation purposes, however, can be granted and revoked freely using the already known shared identity predicates. If a requesting node's public key passes the predicate check, it may obtain a reference to the shared value via the MPC store's get-function. The reference can subsequently be included in private smart contract code, much in the same way as references to DHT entries.

11.2.3. Computation Model

Clients on an Enigma network formulate their computations as private smart contracts. Contract code is written in a Turing-complete, general-purpose scripting language. While being similar to their traditional counterpart, private contracts offer the added ability to run privacy-preserving computations on secret values. These computations are executed by Enigma's MPC environment. To guarantee the environment's security, it is sufficient

Figure 11.1.: Depiction of Enigma's custom feed-forward network.



that only a small number of worker nodes participate. Large-scale redundancy, as in the case of blockchains, is not needed.

The employed MPC algorithm is mainly based on publicly verifiable SPDZ. Enigma's public ledger acts as the bulletin board mentioned in [13]. Any intermediate proofs that are generated during the execution of an SPDZ instance are stored there. Enigma further deviates from the original SPDZ implementation by requiring that values are secret-shared via SSS instead of simple additive secret-sharing, as was the case originally.

While SPDZ is capable of handling arbitrary additions and multiplications on its own, it lacks the high-level control logic necessary for true general-purpose applications. Private smart contracts can provide these control-structures, such as loops and conditions, in a secure way. Hence, by combining the computation logic of SPDZ with the execution logic of smart contracts, Enigma's MPC environment is able to achieve a qualified form of Turing-completeness. This is sufficient to support the integrated scripting language.

Enigma's developers have put several measures in place to help the system with its overall performance.

1. During what they refer to as a "network reduction step", a sampling algorithm selects the nodes that should run a new MPC instance. The randomized nature of the algorithm ensures that computational loads are evenly distributed across the network.

2. To further reduce the number of required compute nodes, the interpreter sees to it, that intermediate results are aggregated as quickly as possible. It does so by rearranging a smart contract's code into a more optimal order, without changing the final outcome of the computation. By applying the properties of addition and multiplication, these operations can be shaped into a *feed-forward network*. Such a network consists of multiple rounds, each with an addition phase and a subsequent multiplication phase. When these phases are executed in parallel, they quickly shrink the number of intermediate results.

3. MPC multiplications require all participating nodes to exchange information (conf. 10.2). Without adjustments, the communication complexity of Enigma's multiplica-

tions would scale quadratically with the number of participants. To prevent this, the multiplications are carried out by a sequence of hierarchically layered MPC gates, as proposed by [31]. This way the communication complexity can be linearized.

11.2.4. Incentive Scheme

Enigma utilizes monetary incentives to secure those parts of its protocol, which cannot be secured with cryptography. To drive network participation, worker nodes are adequately compensated for their services. Worker nodes in the context of Enigma are either nodes offering their storage to hold chunks of the DHT, or the ones lending their compute power during an MPC execution. The necessary funds are collected from transaction fees paid by the clients.

Compute nodes are further required to make security deposits prior to an MPC round. This is an important step in overcoming MPC's inherent fairness problem. Without security deposits, the MPC participants have nothing at stake, and a malicious node is free to try and tamper with the protocol at will. Potential attack vectors include the injection of forged values or DoS attacks, where, once the computation's result is learned, the node refuses to forward it to its peers.

Such attacks are impossible to prevent directly, but they can be reliable discovered during SPDZ's public verification phase. In the case an attempt is detected, the malicious node's security deposit is confiscated and it is excluded from future MPC rounds. This punishment is sufficient to acts as a strong deterrent for any attacker.

Part V.

Evaluation

12. Method

In the previous part of this work, we identified three different techniques for verifiable computation: *trusted oracles*, *zero-knowledge proofs* and *multi-party computation*. We looked at each technology in isolation and gave a detailed description of their technical backgrounds and current state-of-the-art.

In this chapter, we apply these technologies to our initial use case of outsourcing computations in energy-related blockchain systems. At first we take a look at each category individually and evaluate the performance of prominent implementations. Then, we compare the techniques amongst each other and highlight their different advantages and weaknesses. This helps us determine which technology is best suited for our prototypical use case implementation.

Because the energy industry is considered critical infrastructure, applications in this space have special requirements that go beyond simple performance metrics. We therefore construct a theoretical framework that respects these conditions. This way, we are able to compare each technology based on a list of objective criteria.

Our framework is an adapted version of [101]. We have changed the model to better fit our verifiable computation use case. In total, it consists of nine criteria belonging to three different categories. They are:

- Security
  - **I**ntegrity: can correct execution of computations be guaranteed?
  - **T**ransparency: is the process traceable for outside observers?
  - **C**onfidentiality: is user data kept secret?
  - **P**rivacy: are the identities of the users protected?
- Performance
  - **T**ransaction **S**peed: how fast can a single computation be executed?
  - **M**emory **C**onsumption: is the limited space of blockchains a constraining factor?
- Practicality
  - **M**aturity: is the technology in a practice-ready state?
  - **U**sability: how accessible is the system for users and developers?
  - **E**xtensibility: how easy is it to add new functionality?

For evaluation, we rely on a traditional grade system with five different scores. The possible grades are *excellent (++)*, *good (+)*, *average (0)*, *fair (-)* and *poor (--)*. For each category, we also justify our grading in prose.

13. Inter-Technology Comparison

We continue by reviewing each technology in isolation. This is done by drawing comparisons between different implementations which use said technology as their foundation. Our evaluation is based on similar meta studies and insights won from the respective architecture descriptions.

13.1. Trusted Oracles

	Security				Performance		Practicality		
	I	*T*	*C*	*P*	*TS*	*MC*	*M*	*U*	*E*
IOC	-	++	--	0	0	++	++	++	++
SGX	0	--	++	0	++	++	++	+	0

Table 13.1.: Verifiable computing potential of select trusted oracles techniques.

Security [71] conducts an integrity analysis for a number of current oracle implementations. The authors devise a "reliability score" derived from fault tree diagrams and resulting reliability equations. Applying these equations to the oracle architectures eventually yields a score restricted to an interval between 0 and 1. The closer this number is to 1, the lower the chance of an oracle to yield an incorrect answer. In the result, all oracles achieved scores between 0.99 and 0.93. While this initially seems like an overall high level of correctness, realistically, a failure rate of 7% is unacceptable considering our system should be trusted in the same way the consensus algorithm itself is trusted.

The reliability scores were not influenced by whether IOC or SGX was used as the underlying technology, even for oracles with scores close to 1. This is noteworthy, because recent literature has successfully shown multiple different attacks on SGX [30, 96, 95]. While its cryptographic foundation is sound, the TEE is vulnerable to side-channel attacks that make use of hardware-specific characteristics. Especially timing-based attacks on SGX's cache infrastructure have seen some success in realistic scenarios. While the possibility of such attacks has been considered during the reliability analysis, gauging their real impact remains hard due to their novelty. Further research in this department is required.

When it comes to transparency, we find great differences between IOC and SGX. Because most IOC oracles exist fully as smart contracts on the blockchain, their execution paths are fully traceable by an outsider. This highly transparent design is a vital part of the original blockchain idea and leads to a strong notion of trust in the system. SGX, on the other hand, requires great trust in the manufacturer as a third party. While the employed cryptographic primitives are open-source, they rely on several opaque services to function correctly (e.g. attestation, EPID, etc.). This is further amplified by the black box design

of Intel's processors. Trusting an SGX-enabled system therefore always includes trust in the manufacturers honest behavior, a requirements which might be too steep for some high-security use cases.

For preserving data confidentiality, however, these very architectural considerations lead to opposite results. Due to the public nature of IOC oracles, all of their internal data is also made public. In our research, we did not encounter a pure IOC approach which allowed users to encrypt their private data prior to the computation. Indeed, this would be difficult to implement, since traditional public key infrastructures require fixed user identities to bind their keys, something which is not possible in the IOC model. Intel circumvents this problem by assigning a permanent hardware identity to each SGX device. Encryption keys are tied to this identity via EPID (conf. 4.2). By encrypting sensitive data with these keys, users can keep their secrets private and still obtain valid computation results.

In terms of privacy, both types of oracles exhibit similar behavior. Users interact with IOC oracles through simple smart contracts front ends with no additional security layers. This means that they are subject to the same pseudonymous privacy model as the blockchain itself. Most importantly, even though real identities are hidden behind public identifiers, these identifiers stay the same for each and every transaction. As such, the real identity of an active user can be revealed by correlating their computation requests. Users of TEE oracles are subject to the same restrictions but also face the added challenge of holding uniquely identifying hardware keys. For SGX specifically, this problem is solved with Intel's EPID group signature scheme, which hides a device's key among multiple foreign keys.

Performance Clear differences between the two oracle technologies also become apparent with respect to transaction speed. Namely, IOC constructions are redundant by design. Each computation result must first be filtered and aggregated before obtaining the final answer. The inherent communication delay between the compute nodes leads to a performance bottleneck which cannot be remedied with faster hardware. SGX oracles, on the other hand, require only a single node to execute the computation, eliminating any redundancy in their system. Additionally, enclave execution is directly supported by hardware and not necessarily slower than computations in the regular host environment [33]. Because of this, IOC oracles generally exhibit much slower query times than their SGX counterparts [71].

An increase in blockchain memory consumption could not be detected for any of the oracle variants. IOC does not introduce any new cryptographic primitives with additional space requirements at all. SGX allows encryption of transaction payloads but this does not increase the message size and neither does key management, since it is fully handled off-chain.

Practicality Blockchain oracles in general are one of the oldest applications of the technology. Consequently, most implementations have reached a high level of maturity. There exist several production-ready examples for each of the listed approaches, e.g. Chainlink, TownCrier, TrueBit, etc. (conf. 5). All of these are developed by an active community and have seen use in real-world applications. Thus, they are a safe choice for new projects,

since the chance of them no longer being maintained is quite low.

Similar things can be said about the usability of blockchain oracles, although with slight reservations. From a user perspective, most oracles are perceived as traditional smart contracts and can be interacted with as such. To deliver a computation result, they simply inject it into the user contract's callback function.

IOC oracles at least, provide a similarly straight-forward experience for developers. Since computations happen in an off-chain environment with no additional security precautions, any traditional software stack can be used to create new oracles. In contrast, SGX oracles require the use of Intel's dedicated APIs[1] to access their built-in security features. Using external libraries, however, has become a standard practice in modern software engineering and should not pose a great hurdle for developers.

Since IOC oracles run their computations in traditional software environments, their possible use cases are only limited by the restrictions of general software development. This gives oracle providers great freedom of choice in the services they might offer. While SGX also runs as part of a traditional application, the security features themselves are subject to certain constraints, such as limited clock access (conf. 4.2). This places some limitations on the possible number of use cases. Nonetheless, any pure computation task remains fully supported by the platform.

13.2. Zero-Knowledge Proofs

	Security				**Performance**		**Practicality**		
	I	*T*	*C*	*P*	*TS*	*MC*	*M*	*U*	*E*
Prepocessing	+	++	0	0	0	+	0	+	-
Transparent	++	++	0	0	0	--	-	0	-

Table 13.2.: Verifiable computing potential of select zero-knowledge proof techniques.

Security All modern zero-knowledge proofs exhibit similar integrity guarantees. This is because they are based on the same theoretical framework. More specifically, the zkSNARK definition states, that an adversary's chance of forging a proof must be negligible (conf. 7.1). In practice, this means that the probability of a successful forgery shrinks superpolynomially with an increase in argument size. This basic requirement is met by all implementations we examined. As a result, their proofs have the ability to provide users with a strong notion of computational integrity.

The same can be said about our other security-related evaluation criteria. As is the case with most modern cryptographic primitives, the construction process of all zkSNARKs is fully open-source. Because the presented mathematical techniques are freely available, anyone can verify their correctness. In fact, this concept of complete transparency is vital for a user's trust in the system.

As part of this radical transparency approach, the constructions also reveal the zero-knowledge techniques which they employ for data confidentiality. The original zkSNARK

[1] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html

definition has strict requirements for this property, i.e. no secret is revealed at any point before, during or after the computation. Again, this condition is fulfilled by all our implementations, albeit with a small caveat. Even though no outside observers may extract a user's private data from a proof, any real-world application still requires this data to be entrusted to a compute node. This is necessary, because the compute node needs to include this data during proof generation. Therefore, any zkSNARK-based application that handles confidential data must have access to at least one such trusted node.

For the privacy aspect of zero-knowledge proofs, the usual shortcomings of blockchains have to be considered (pseudonimity, etc.). In the case of confidential information, however, additional restrictions apply. Because secret data cannot be extracted from a proof, the provers employ a special technique which shows that they included the correct information. Initially, any user taking part in the protocol publishes a hash of their secret on the blockchain. As part of the outsourced computation, the prover now recalculates these hashes and includes them in the output. Because forging the result of hash functions is computationally infeasible, this allows users to verify the correctness of the secret inputs. It, however, also allows anyone to observe a user's participation in a specific computation. As is the case with oracle-based off-chaining, this may lead to at least a partial reconstruction of the user's identity based on their activity.

While it seems from the above description, that all zkSNARK schemes guarantee a relatively high level of security, this only holds true in a theoretical setting. If we also consider attacks that do not target the underlying construction directly, e.g. social engineering attacks, the situation drastically changes. Preprocessing zkSNARKs require a trusted setup to generate a CRS ensuring the protocol's correctness [16]. In doing so, these schemes effectively concentrate all trust in their setup phase, resulting in a single-point-of-failure which is highly vulnerable. Anyone able to subvert the third party carrying out this setup could potentially forge arbitrary proofs. Consequently, a preprocessing zkSNARK is only as secure as its setup phase. This problem is completely circumvented by transparent zkSNARKs as they lack this setup. Transparent zkSNARKs thus remain secure, even in realistic scenarios, where one or more nodes might be corrupted.

Performance With constant verification complexity and a small proof size of just 128 bytes, the initial zkSNARK construction of [44] achieves decent on-chain performance at the cost of expensive proof generation. Instead, its main shortcoming is its trusted setup phase which we already discussed. To reiterate, it requires the presence of a third party to generate various computation parameters. Transparent zkSNARK schemes were developed to eliminate the need for a trusted setup and consequently a third party whose corruption could threaten the security of the protocol. This comes at the price of performance that is often much worse than the current state-of-the-art outlined in [50].

[88] includes a detailed comparison of recent zkSNARK implementations without trusted setup. The authors found that most schemes have either linear or logarithmic verification time with respect to input length. Proof generation times, on the other hand, have remained relatively stable with poly-logarithmic complexity across all reviewed implementations. While these limitations might seem a small price to pay for transparent setups, the proof size of these schemes is also massively inflated to the point of impracticality. For most schemes sizes of several hundred kilobytes are common but some even reach up-

wards of tens of megabytes. Considering the current block size limits of popular reference chains (1MB for BitCoin, ∼30kb for Ethereum), this makes these implementations clearly unsuitable for blockchain applications.

Practicality Preprocessing zkSNARKs have recently reached a level of maturity where first trial runs for test projects seem realistic. The most complete zkSNARK library to date, libsnark, currently supports the constructions outlined in [44, 17, 50] and several others. As we have seen, they are actively being used in Zcash and Ethereum's ZK rollup protocol where they provide significant contributions to privacy and performance.

Transparent zkSNARKs, on the other hand, are a much younger technology and still in their initial stages of academic research. As such, their applications are still rather limited. Whereas some show promising results, many new schemes do not even have a reference implementation yet [103, 23]. On top of that, their immense space requirements make them impractical for use with current blockchains.

A similar pattern can be observed for general usability. All of the libraries and tool chains we discussed in 8 are implemented on top of preprocessing zkSNARKs. No such universal tooling currently exists for any of the transparent variants. This includes proof generators, verification systems, smart contracts, etc. Fortunately, however, transparent zkSNARKs use the same R1CS format to represent their arithmetic circuits. Therefore, at least the existing circuit compilers can be reused.

All general-purpose zkSNARK schemes operate on the same computational model and are capable of running the same programs. This includes any bounded algorithm which can be represented as an arithmetic circuit. As it stands, no constructions for complexity classes beyond this currently exist. This relegates zkSNARKs to purely mathematical tasks that do not make use of any hardware specific features, e.g. a node's network stack. While this somewhat limits their amount of possible use cases, they still remain useful for verifying many numeric problems, such as scientific calculations or payment schemes.

13.3. Multi-Party Computation

	Security				**Performance**		**Practicality**		
	I	*T*	*C*	*P*	*TS*	*MC*	*M*	*U*	*E*
SPDZ	++	++	++	+	--	+	-	-	-

Table 13.3.: Verifiable computing potential of select MPC techniques.

Security In regards to the security of MPC algorithms, there exist a number of different models which are mostly defined by the behavior of a potential attacker. Three of these models that reoccur throughout literature are mentioned in [105]:

1. Semi-Honest

   An adversary is not allowed to deviate from the original protocol. Their only motivation is the deciphering of secret data. This is the least restrictive model and most algorithms are proven to be secure in it.

2. Malicious

   In the malicious adversary model, the attacker's motivation remains the leaking of private secrets. They, however, have the ability to disrupt the protocol in various ways, e.g. by attempting to forge ciphertexts or publish incorrect information. Whenever the attacker deviates from the protocol in this way, it is noticed immediately by the honest participants.

3. Covert

   A covert adversary acts much in the same way as a malicious one but additionally has a chance to stay unnoticed when they disrupt the protocol. In other words, the attacker is essentially free to manipulate the protocol at will. Hence, this adversarial model is the most realistic and poses the greatest challenge to a secure algorithm design.

The authors of [35] claim their SPDZ construction guarantees the integrity and confidentiality of outsourced computations even in the case where $n-1$ of the n participants act as covert adversaries. This is an exceptionally high level of security, since it basically allows anyone to make use of the protocol without having to trust the rest of the network.

Just like zkSNARKs, most MPC algorithms are designed to be used as cryptographic building blocks with a heavy background in academics. As such, their architectures are well-known and detailed technical descriptions are easy to find. SPDZ and its variants in particular feature extensive documentation in their whitepapers and online. On top of that, reference implementations are freely available and open-source[2].

This combination of factors leads to complete user transparency. No step of a SPDZ round requires the involvement of a trusted third party that hides certain details from its clients. Instead, all computation artifacts are made publicly available on the blockchain for anyone to verify. In fact, we already mentioned above that users do not even need to trust each other, since the employed MAC scheme is secure in the case of $n-1$ covert adversaries. This makes SPDZ a completely trustless protocol.

As is the case with the other discussed technologies, the usual pseudonymity restrictions of blockchains apply in the absence of additional anonymization methods. Nonetheless, SPDZ offers a few advantages over them when it comes to correlation attacks on user requests. As it stands, current SPDZ protocols require the participation of all network nodes [90]. While this type of redundancy is often criticized as detrimental to the overall performance of SPDZ, it also serves to increase user privacy. Whereas the other verifiable computing techniques often have a clear initiator, SPDZ only runs once for all nodes. The secret-sharing algorithm makes it virtually impossible to distinguish between nodes who actually took part in the computation and the ones who did not. Albeit weak, this adds a layer of privacy to the protocol which is lacking from the other examined techniques.

Performance One of the SPDZ protocol's core tenets is its separation into two distinct phases. Actual computations are processed during an online phase under collaboration of all nodes. This phase is very efficient and runs in quasi-linear time. The real performance bottleneck stems from the preceding offline phase, which is responsible for initializing

[2] https://github.com/bristolcrypto/SPDZ-2

secret values. Improving the runtime of this preprocessing step has been the topic of much recent literature.

In a 2013 article [34], the same authors introduce SPDZ2 as an enhanced versions of their original protocol. By slightly adjusting the algorithm's theoretical framework, they were able to achieve better performance across the board. For both, the online and the offline phase, the running times could be decreased by a factor of two. Following this initial improvement came multiple further adjustments to the offline phase specifically. Overdrive [65] and TopGear [12] are just the names of two such technologies. They use lattice-based homomorphic encryption to speed up parameter generation during preprocessing. Since this is evidently the most expensive step of SPDZ, this leads to renewed large performance gains for the algorithm.

Publicly verifiable SPDZ requires several items to be published to the blockchain. Most importantly, these include each participant's Pedersen commitment of their secret input, but also the final solution and any intermediate result obtained during computation. Fortunately, all of these values are relatively small and do not place excessive restrictions on the choice of an underlying blockchain implementation.

In the case of overly large inputs, Enigma has provided an elegant workaround by placing these outside the chain. When stored in a DHT for example, these values can still be accessed during the computation via their respective hash references. These hashes are tiny in comparison to the actual data and easily fit into a single block. This way, outside data can be efficiently included in SPDZ calculations without overburdening the blockchain.

Practicality Let us begin by pointing out that efficient MPC protocols are a relatively young technology. Despite their strong security guarantees, they have received far less attention by the blockchain community than other protocols for verifiable computing, e.g. zkSNARKs. As such, real-world implementations are hard to come by. In fact, SPDZ was the only MPC algorithm which we found, that has reached a somewhat acceptable level of maturity. It is under constant development with new and improved variants being released frequently (see above).

Nonetheless, SPDZ remains a niche technology. Even its newer incarnations show poor performance, making them unsuitable for more expensive computations, such as large-scale scientific calculations. As a result, even the most prominent implementation – Enigma – has switched to zkSNARKs as its underlying technology in more recent builds [40]. Currently, the reference implementation for SPDZ and its various improvements can be found in the SCALE-MAMBA framework[3]. It is, however, of a more experimental nature and not tailored to blockchain-specific use cases.

Another reason for the reluctant community adoption of SPDZ are the great strides made recently in the development of fully homomorphic encryption schemes. While still not practical, several revised designs [47, 21, 28] have lead researchers to believe that FHE schemes for general-purpose applications are probable if not realistic in the future. Additional interest is garnered by the fact, that besides providing the same security guarantees as SPDZ, they also span a much wider range of potential use cases. This is because the verifiable computation framework which they provide is not restricted to blockchains, but may be applied to other popular fields, e.g. machine learning on private data [61].

[3] https://github.com/KULeuven-COSIC/SCALE-MAMBA

SPDZ works on the same arithmetic circuit abstraction as zkSNARKs. This means that developers proficient in writing these circuits can expect a certain level of familiarity. Enigma, for example, allows its private contracts to be written in high-level languages such as Rust. Other implementations, however, do not support the use of intermediate formats, namely R1CS. Therefore, all their programs have to be compiled from a raw circuit description language such as VHDL.

Due to their shared dependence on arithmetic circuits, the extensibility of SPDZ is analogous to its zkSNARK counterpart. As long as a program can be expressed as a circuit, it can be safely executed by the protocol. This is true for all algorithms with a fixed upper bound. Again, algorithms beyond this complexity class are not covered. Hence, only relatively simple and purely mathematical calculations may be evaluated and verified with SPDZ.

14. Use Case Suitability

After evaluating each of the presented technologies in isolation, we proceed by determining which is most suitable for our use case at hand. We do this by taking into consideration the specialized needs of energy-related applications and applying them to our findings up to this point. Finally, we choose the most promising technology based on our evaluation criteria and make it the foundation of our prototype implementation.

Security/Practicality Trade-Off From the above analysis, a clear dichotomy between security and practicality becomes apparent. This observation should come as no surprise, since newer solutions always try to improve on the currently accepted status quo. Incidentally, we have listed the technologies in increasing order of security. As such, trusted blockchain oracles offer the least amount of security guarantees but they have also reached the highest level of maturity. Especially IOC-based oracles have been in use in real-world applications for several years now, proving their overall efficiency.

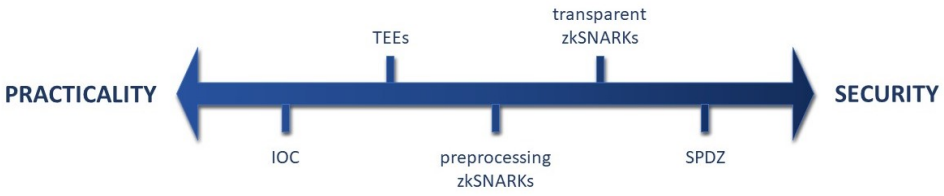
SGX-based variants offer a little more in the way of trusted execution at similar levels of availability. Their security promises, however, can often only be guaranteed in theory. We were able to identify a number of shortcomings which severely impact applicability in more realistic settings. Primarily, these include the CPU vendor as a necessary trust anchor and the existence of multiple compromising side-channel attacks.

On the other end of the spectrum, we find the SPDZ MPC algorithm and its more recent extensions. At least in theory, this technology offers unparalleled security. Not only does it protect the integrity of the computation, it also completely hides any private input data on top of also obfuscating the real identities of its users. As we have mentioned, however, the protocol is still very immature. Even though its original version was invented in 2013, around the same time as zkSNARKs, a more widespread adoption has failed to materialize. Despite active development in recent years, performance remains poor and the number of production-ready implementations low.

Somewhat of a middle ground between speed and usability is offered by the zkSNARK technology. All examined implementations offer good security properties and high transparency. The need for a trusted setup remains a real vulnerability concern but the community is acutely aware of it and several solutions are being developed. Performance is somewhat lackluster but steadily improving. More established zkSNARK variants already exhibit very fast verification speeds. Instead, the main research focus currently lies in better proof generation times and a decrease in setup costs.

Choosing the Right Approach Let us now apply our findings to applications directly related to the energy industry. Incentive-driven blockchain oracles are already available today and support the broadest spectrum of use cases due to their unconstrained execution model. On the other hand, their comparatively high failure rates make them undesirable

Figure 14.1.: Illustration of the security/practicality trade-off problem.



for our chosen field, namely the energy industry. It is often seen as critical infrastructure where even a single outage might have lasting financial and social effects [100].

On top of that, distributed oracles largely lack support for secret user data. In systems that are intended for use by large swathes of the population, preserving confidentiality is mission-critical. Transmitting information like energy consumption, location or payments in the clear gives attackers an easy target. Correlating these data points could potentially allow them to stage large scale attacks on the privacy of individual users.

Oracles based on the Intel SGX platform initially seem like a good fit for the energy industry. They provide decent security and an extensible execution model which can be molded to a multitude of use cases. Their biggest shortcoming, the reliance on the vendor as a trusted third party, is mitigated by the fact that most energy industrial systems are already operated by third parties that need to be trusted, e.g. governmental facilities. A tight cooperation between said government and Intel as the vendor can resolve some of these trust issues as the necessary trust is spread evenly among the participants. In this case, users who, for example, trusts only one of the parties, extends their trust to the other as part of this cooperation. As a result, we can already see a few real-world examples for applications running on SGX [22, 8].

For our setting, however, we try to focus on technologies that require even less trust on the part of its users. This becomes important in situations where none of the system operators are trusted. One such scenario might be P2P electricity markets with the energy providers as the sole operators. Since they concentrate all the power, these providers still have the ability to manipulate the market in their favor, even if the mentioned security measures are in place [70]. By conspiring with TEE vendors, for example, calculation results could be forged and prices artificially inflated.

This can be prevented with verifiable computation techniques that only rely on cryptography for their security guarantees. As long as these guarantees hold, even powerful adversaries, such as grid operators, do not have the ability to tamper with the system. One such technique is MPC. By providing a completely transparent and trustless environment, the effects of power centralization are dissolved. User data is kept private with advanced secret-sharing techniques. All in all, it can be said, that MPC has exhibit all the security properties, which are desirable for applications in the energy industry.

Unfortunately for us, its poor performance proves to be a real hindrance. Most energy-related applications deal with intricate large-scale systems containing numerous actors and variables. Thus, they require high performance and overall throughput to guarantee timely results [95]. We believe that at this point, current MPC implementations do not satisfy this requirement and with the hesitant adoption of this technique, this is unlikely

to change in the near future.

On the contrary, we find that the balance between security and performance struck by zkSNARKs offers the most promising solution for our problem. Also based on the principles of cryptography, they exhibit many of the same security guarantees as MPC. Indeed, all zkSNARK constructions operate transparently and mostly trustless. Unlike MPC, however, private user data has to be shared with the compute node. While this is undesirable for many confidential applications, it has limited impact on the practicality of energy-related uses cases. This is, because most of the time, any sensitive data is only related to the production of energy itself, that is to say, data that is usually already available to the operators of the compute nodes. To reiterate on the P2P electricity market example, users would have to give up their energy consumption and bids to calculate a market clearing price. Since the compute nodes are operated by the energy providers in this area, they already know this information – no confidential data is leaked.

Most importantly, however, zkSNARKs are the center of many ongoing research projects (conf. 7.1). This leads us to believe that the technology is likely future-proof. In the short time since their inception, many shortcomings have already been addressed. This includes the trusted setup requirement as well as their middling performance. Since development is backed by many significant players in the blockchain space, we expect this trend to continue. For these reasons, we choose zkSNARKs as the preferred technological foundation for our prototype.

Part VI.

Implementation

15. Applications of Verifiable Computing

Before discussing our actual implementation, we want to give a short summary of the real-world examples which have motivated this work. We start by providing an overview of the currently researched blockchain use cases in the energy industry and their relative importance. We then proceed by highlighting certain commonalities and derive a solution approach which is able to cover a significant portion of the relevant use cases. This common approach later serves as the foundation for our blockchain-based implementation.

15.1. Use Cases in the Energy Industry

The energy sector is highly critical infrastructure and plays an essential role in the well-being of developed nations. Naturally, large efforts are made towards securing its most vulnerable elements against foreign attackers and outside manipulation. In many cases blockchain and public ledger technologies seem like a natural fit. Their distributed nature significantly improves the resilience of critical systems, while at the same time, their fully transparent design promises a new level of control for consumers.

Andoni et. al. [7] give a comprehensive overview of the current state of blockchain use cases in the energy sector, breaking down individual projects into their components. The authors identify *P2P energy trading* as the primary driving force behind blockchain adoption with one third of all projects utilizing it in some form. To summarize, in this idealized form of energy trading, every consumer connected to the grid can also divert locally generated power (e.g. through photovoltaics) back into the grid and receive monetary rewards. Instead of a central grid operator managing this distribution of power, all participants are connected and trade their energy directly. In the proposed setting, this is enabled by secure smart-meters and a blockchain-based auction platform. With these tools, communities would become completely independent from commercial energy suppliers as they could produce and distribute their own energy locally.

Other energy-specific use cases include *asset management* (11%), *metering & billing* (9%), *grid management* (8%), or *green certificate trading* (7%). A specific problem that has received renewed attention with the advent of blockchains is that of *Optimal Power Flow (OPF)*, a specific sub-category of grid management. OPF deals with the question of optimal energy distribution in a network with a fixed number of producers and consumers and several other constraining factors. The technique itself is based on multiple long standing mathematical models [66, 63, 104], but has recently been adapted to decentralized environments. Several works suggest the use of public ledgers as a control mechanism for the optimization algorithm, so as to eliminate the need for a trusted third party [76, 4]. This approach appears especially promising in the context of P2P energy trading where no central control entity exists but energy flow still needs to be optimized [91].

15.2. Linear Programming

All of the listed use cases share the fact, that they depend on some form of optimization algorithm as part of their architecture in order to be able to fulfill their desired objective. The various different OPF models in particular have already been formulated as mathematical optimization problems, but the same can be done for our other candidates.

In the case of P2P trading, for instance, we are faced by the central question of *market clearing*. In a cleared market, the purchase price of the offered product is set so that supply and demand are allowed to meet, but deriving the exact market clearing price can be a difficult task, depending on the number of confounding factors. To solve this, researchers have brought forth a number of numeric optimizations models that seek to provide market clearing mechanisms that are much more efficient than the traditional auction-based heuristics [11, 58]. Similar optimization models also exist for the trading of green energy certificates [20] and the management of energy generation assets [25].

In practice, all of these optimization problems can be formulated as *linear programs (LPs)*. Linear programming is a sub-category of mathematical optimization from operations research, where the problem's constraints are presented as equalities and inequalities of purely linear terms. Originally intended for maximizing the yield of factories, the aim of linear programming is to find the optimal production settings which maximize revenue while also respecting the resource limitations for each manufactured product. Through input transformation, this method can be applied to any problem where sparse resources must be allocated in a way that maximizes a certain objective function. Consequently, a generalized linear program is given by its canonical form

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ \text{and} \quad & \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

where $\mathbf{x}$ is a vector containing the so-called *decision variables*, which represent the optimal input values after the program has been solved. Secondly, the vector $\mathbf{b}$ represents the total amount available of each required resource, constraining the possible values for each decision variable. The matrix A, on the other hand, contains each decision variable's coefficient in the different resource equations, where a higher coefficient means that the variable requires a larger share of the specific resource. Lastly, the vector $\mathbf{c}$ represents the decision variable's contribution to the targeted objective function, or the total revenue in the original setting.

The solving of linear programs is an ever-evolving field with a substantial commercial interest and over the years many novel solution approaches have emerged. One of the earliest linear programming techniques was developed by George Dantzig in 1947 [37]. Dubbed the *simplex algorithm*, this method adopts insights from geometry to model a problem's solution space as a convex polytope. To find the optimal solution, the shape is defined such that each of its corners represents a feasible assignment for the decision variables. By starting at the polytope's origin and traversing its corners along the edges, the algorithm is able to eventually reach an optimal solution.

At the time of its discovery, the simplex algorithm revolutionized the field of mathematical optimization fundamentally, as it was able solve previously unsolvable problems

rather efficiently. Since then, the algorithm has been adopted and improved in numerous different software frameworks. With its good performance and ease of use, it continues to form the basis of many commercial linear solvers, such as IBM ILOG CPLEX[1] or Gurobi[2], despite the fact that newer, more efficient methods have already been developed [64, 43].

[1] https://www.ibm.com/products/ilog-cplex-optimization-studio
[2] https://www.gurobi.com/

16. Implementation Details & Challenges

Similar to existing LP solvers, we have chosen the simplex algorithm as the foundation for our implementation, mainly due to its efficiency and usability. We implement the algorithm as a zkSNARK so that the private user data can be protected while at the same time guaranteeing correctness of the optimization results. For this, we choose the ZoKrates framework discussed in 8.2. ZoKrates provides us with an expressive language to formulate the optimization algorithm as well as the necessary tools to publish and verify the proofs using a blockchain platform.

Implementing the simplex method for traditional computer architectures is a relatively well-known problem. Over the years, multiple improvements over the original algorithm definition have been devised, which further increase its performance on such platforms [51, 57].

To our knowledge, however, there are currently no simplex implementations which operate solely on arithmetic circuits. This is noteworthy, as the limited computation model of these circuits would impose new restrictions that have to be accounted for. Due to a lack of alternatives, we therefore implement the algorithm from scratch, using Dantzig's original tableau method of selecting and exchanging pivot elements. Our exact code can be found in appendix A. In the following, we will instead include a high-level discussion of the biggest challenges we encountered during our implementation phase and how we solved them.

16.1. Runtime Complexity

Like all other circuit compilers based on R1CS, ZoKrates is subject to the limitations of the format's computational model. Algorithms, which would normally be straight-froward to implement on a von Neumann architecture, must be altered to fit this new model.

The biggest challenge here is overcoming the bounded control flow restrictions of arithmetic circuits. In general, all arithmetic circuits have a fixed size and therefore finite runtime. Consequently, useful programming constructs like input-dependent loops and recursions are not allowed by ZoKrates (or any other R1CS compiler), since the compiler is unable to predetermine their exact runtime. Because such constructions could potentially repeat ad infinitum, this makes it impossible to unroll them into a fixed-size circuit layout.

In practice, this restriction is often circumvented by disallowing recursion and giving a static upper bound to loops. ZoKrates follows the same approach, by requiring that all loops run a fixed number of times. Additionally, all iterations must indeed be executed and cannot be skipped over with conditional statements, etc. This behavior is enforced in code by only allowing one type of looping construct: a modified version of the traditional *for-loop*. The index variable is always a single field element of the underlying zkSNARK construction and its range is defined by compile-time constants, which are also field el-

ements. Knowing the maximum amount of iterations at compile-time ensures that the compiler can successfully unroll the loop.

This peculiarity of arithmetic circuits has severe implications for our simplex algorithm. For example, a worst-case instance of the algorithm has to make one iteration for each decision variable in the tableau to reach an optimal solution. The famed efficiency of the algorithm only comes from the fact, that typically this time can be cut short by skipping some of the variables. Therefore, in the average case, the simplex algorithm terminates in cubic time with respect to its input length. However, because ZoKrates requires our main execution loop to have a fixed duration, we cannot utilize this shortcut and terminate sooner. This leads our algorithm to always have worst-case performance. Even if we have reached an optimum early, the remaining iterations of the loop must be taken.

16.2. Numeric Calculations

ZoKrates provides two data types natively – finite field elements and Booleans. To reiterate, an element of a finite field can be any positive integer up to the particular field modulus, which, in our case, is specified by the zkSNARK construction. The Booleans in ZoKrates are implemented as simple field elements restricted to the values one and zero. Linear programs, on the other hand, need to operate on values which often represent naturally occurring circumstances. As such, they are usually represented by the set of real numbers $\mathbb{R}$. The simplex method in particular requires frequent divisions with fractional results. Unfortunately, ZoKrates' provided data types are not equipped to handle such operations.

Finite fields support the same four basic arithmetic operations as all other fields. Addition, subtraction and multiplication behave as expected with the only caveat that results be reduced by the field modulus. Division in a finite field, however, can lead to unexpected results, since the multiplicative inverse of a field element a is defined as the number a^{-1}, with $a * a^{-1} = 1 \ (mod\ p)$, where p is the field modulus. Consequently, in the case, where our quotient would be a fraction, we instead obtain another field element which satisfies the above equation. Therefore, this definition of division does not have the properties which are necessary for the sort of numerical calculations we require for our simplex implementation. We are thus forced to create our own numeric data type that can accurately represent divisions in the reals. Several approaches were considered:

- Floating-Point Numbers

  Floating-point numbers are the most common way to represent reals in modern computer systems [60]. Specialized hardware such as dedicated floating-point units and general-purpose GPUs have made their performance highly efficient. They have a fixed size and rely on simple bit-level manipulation to implement arithmetic operations. Their memory is separated into three parts, the *sign*, the *mantissa* and the *exponent*. Depending on the chosen bit-width, this allows accurate representation of large decimal numbers with very little loss of precision.

  While ZoKrates theoretically allows simulating bit-fields with Boolean arrays, this technique incurs a heavy performance penalty. Specifically, for every bit-wise operation, all Boolean variables in the array must be modified. Not only is this approach

much slower than comparable field operations, it also increases the circuit size substantially by introducing a large numbers of additional gates. Unlike regular computers, there currently exists no specialized hardware which could potentially speed up these operations. For these reasons, we quickly dismissed floating-points as the solution for our problem.

- Fixed-Point Numbers

  Fixed-point numbers are often used as an abstraction for reals in systems where floating-points are not an option [99]. This is normally the case in embedded devices which lack the necessary hardware to speed up floating-point computations. They are stored as traditional integers along with a scaling factor, usually a power of ten. To obtain a fixed-point's real value, we simply multiply the integer with its scaling factor.

  While fixed-point numbers are certainly a better candidate for our implementation than floats, they still exhibit some shortcomings which make them unusable. Namely, during fixed-point operations, the scaling factors of both operands must match. This can be done by either multiplying or dividing the underlying integers and adjusting their scaling factors accordingly. This requirement is problematic in our setting for two reasons:

  Depending on which type of integer division is used, small rounding errors may be introduced. Unfortunately however, the simplex algorithm is very sensitive to such rounding errors, as it contains many exact comparison operations. Therefore, as is the case with floating-points, rounding errors make fixed-point numbers unsuitable for our calculations.

  Secondly, ZoKrates does not natively support any kind of integer division. Again, such behavior could be simulated by implementing integers as Boolean arrays, but this would lead to heavy performance losses. Since we did not find a practical way to overcome the absence of a division operation, we have therefore decided against using fixed-point arithmetic.

- Rational Numbers

  Because the mentioned implementations of real numbers have proven insufficient for the listed reasons, we decided to implement them using our own custom rational number type. Rationals are a good approximation for real numbers in the context of linear programming. Real-world problems rarely use values which lie in $\mathbb{R} \setminus \mathbb{Q}$ and thus cannot be represented by our type. Even if this were the case, any introduced offset would only affect the final outcome of the optimization negligibly.

  Rational numbers are a good fit for ZoKrates' existing type system. They are easily implemented using a Boolean for the sign and two field elements for numerator and denominator. Operations do not require expensive bit-level logic but are instead simulated with a few simple conditional checks and the existing field operators. Even the previously problematic division operation is reduced to a rather straight-forward multiplication where the numerator and denominator of the multiplier are switched. An additional benefit of dividing two rationals in this way is that no rounding er-

rors are introduced. Thus, the simplex method's precise comparison checks are left unaffected and cannot lead to false optimization results.

As an important implementation detail, we never reduce the results of arithmetic operations on our rational numbers. Doing so, would require finding the greatest common divisor (gcd) of the resulting fraction and adding even a simple gcd algorithm, e.g. Euclid's, to the proof would severely hamper its performance. Additionally, ZoKrates does not even support the required modulus operations out-of-the-box, which would have to be implemented also. Therefore, any values obtained from our optimization program must be reduced off-chain. We feel that this restriction is acceptable in light of the changes which would have to be made to the zkSNARK in order to reduce fractions internally.

16.3. Input Correctness

Recall from 6, that a traditional zero-knowledge proof's only ability is verifying the knowledge of a certain fact. With zkSNARKs we can expand this definition to also include such a provable fact in arbitrary computations. However, in both cases the fact itself remains hidden from outside observers. While this behavior is obviously in the interest of the original creators, it carries a significant implication for verifiable computation schemes that are based on zkSNARKs. This is because, in a VC scheme, the facts which are protected by the zkSNARK are actually the user secrets which act as the inputs to the verified computation. But since the zero-knowledge property also applies to the user which supplied the secret in the first place, there is no way to ensure that the secret is indeed the one used as an input to the computation. A malicious compute node could potentially use any input data in place of the user-submitted data and still produce a valid proof. Such manipulation attempts are undiscoverable by the original user due to the zkSNARK's zero-knowledge property.

We prevent this sort of attack by employing an input hashing scheme. In our scheme, the zkSNARK initially generates hashes of all the optimization inputs and saves them for later use. Once the simplex algorithm finishes, the zkSNARK outputs the input hash values along with the optimization result. This way, in addition to verifying the result, users also have the ability to check that the correct inputs were used. To do so, they simply recompute the hash values locally using their secrets and the same hashing scheme as the zkSNARK and compare them to the hashes obtained from the compute node. Should the values match, the users can be assured that the inputs were the correct ones, otherwise the hash function would be broken. By requiring that users publish their hashes prior to the computation, we can also turn this scheme into a publicly verifiable one. Because of the employed hash function's one-way nature, no user is able to reconstruct the original secrets, yet can easily verify the that everyone's data was correctly included in the optimization.

When implementing any secure hashing scheme, the first instinct is to utilize a member of the SHA family of cryptographic hash functions. While ZoKrates does indeed offer support for SHA512 as part of its standard library, this approach strongly degrades overall performance. All hash functions of the SHA family are specifically designed for conventional computer architectures. As such, they rely heavily on bit-manipulation tech-

niques, e.g. bit-shifts, bit-wise logic operators, etc. As mentioned in our discussion on floating-point numbers, these techniques are quite efficient on modern hardware but perform poorly in arithmetic circuits. In our specific example, using a single invocation of the library-provided SHA512 implementation adds around 50.000 constraints to the circuit alone.

To overcome this limitation we opt for an alternative implementation of a hash function with superior performance characteristics on arithmetic circuits. The MiMC family of hashing functions is one such candidate [3]. As its name implies, MiMC achieves this performance by minimizing the amount of multiplications during a round, an operation which is naturally expensive in arithmetic circuits. Initially designed for use in Circom, its promising benchmark results have lead to quick adoption by competing circuit compilers. ZoKrates has built-in support for the MiMCSponge and MiMC7 algorithms. After experimenting with different parameters, we have chosen MiMC7 with 10 rounds as our hash function as this leads to the least number of additional constraints. A single invocation requires 42 constraints, a significant improvement compared to the 50.000 constraints of SHA. Optionally, the number of rounds can be raised for increased collision resistance. Even at a highly secure 90 rounds, a single hash produces only 362 additional constraints[1].

16.4. Further Considerations

In addition to the points listed above, we encountered several non-critical obstacles throughout the implementation phase. Most of these obstacles are specific to the ZoKrates compiler and may not appear in other circuit compilers. We nonetheless illustrate our solutions so as to clear up any confusions around the code contained in the appendix.

- Missing Constants

  ZoKrates does not support the declaration of constant values at a top-level. This is especially bothersome, considering the fixed circuit size makes constants mandatory in many places. It leads to a plethora of "magic numbers" in the source code, significantly reducing overall readability and maintainability. We use the generic macro pre-processor GNU m4[2] to overcome this limitation. Using m4 allows us to insert top-level constants in our source code and expand them during an initial pre-processing step. This step is fast since it occurs fully locally and makes it possible to parameterize an otherwise static circuit.

- Faulty Arrays & Structs

  Multi-dimensional arrays and object-like structs are one of ZoKrates' touted features, which supposedly give it the look and feel of a regular programming language. Unfortunately, as of version 0.6.2 both are plagued by several bugs and inconsistencies. More specifically, indexing elements, respectively members, in any way that depends on an input variable, will inevitably lead to compiler crashes. This problem is not present in other circuit compilers. Since we still wanted to utilize the extensive ZoKrates tool-chain, a work-around had to be found. We settled on representing all

[1] https://github.com/Zokrates/ZoKrates/pull/593
[2] https://www.gnu.org/software/m4/m4.html

complex values as a collection of one-dimensional arrays of the specific type. Multi-dimensional access can easily be simulated by multiplying the row index with the total number of column elements and adding the column index to the result. This technique can be observed in other environments lacking complex data types, e.g. native machine code.

- Input & Output Helpers

  Inputs and outputs in ZoKrates take the form of single-line JSON strings with unique formation rules. To facilitate interaction with other programs, we have created several helper scripts that form a tool-chain around ZoKrates and process incoming and outgoing data. With this, we are able to accept linear programs in their canonical simplex tableau form. The tableaus can be constructed by hand or contained in a .csv-file. Our script parses the data, generates the appropriate JSON string and passes it on directly to ZoKrates. Once the computation has finished, another script receives ZoKrates' textual output on the other end of the pipeline and makes it human-readable. It prints the solved simplex tableau in its canonical form and lists the hashes obtained from the secret inputs. This way, any inconsistencies can be discovered immediately.

17. Results

After implementing our version of the simplex optimization algorithm as a zkSNARK in ZoKrates, we now take a look at its performance. To do so, we adjust the values of two main components: the number of variables and the number of conditions in the optimization problem. We analyze the results in three different categories and give our interpretation.

As an important side note, since the circuit size is fixed, our algorithm has a deterministic runtime as discussed in 16.1. Therefore we are able to reuse the same optimization problem for all test runs and obtain consistent and comparable result. This would not be possible in a conventional computation model.

All tests were run on a worker node in an isolated environment. The node possesses an Intel Xeon Gold 6152 CPU with 22 Cores, 44 Threads and 30MB of cache. Each core runs at a clock speed of 2.1GHz. The used RAM is a 32GB DDR4 Dual Rank RDIMM module with a clock rate of 2666MHz. Data is read from an SAS SSD with a transfer rate of 12Gbps.

17.1. Performance Measurements

To give an accurate account of our implementation's performance, we have chosen three key evaluation metrics. The results of our analysis can be viewed in fig. 17.1.

To start, our most important criterion is the time it takes to compile a single instance of our optimization circuit. Even before our measurements, we projected this metric to be the implementation's primary bottleneck. We tested this hypothesis by varying the number of decision variables and/or conditions of the simplex algorithm and compiling a brand-new circuit for every possible input combination up to a total of ten-by-ten parameters. The resulting 2D surface map is the first chart shown in figure 17.1. It reveals that the surface area stays comparatively flat for lower combined parameter counts but quickly experiences a drastic spike in compile time if we increase both, the decision variables and the conditions simultaneously. Whereas creating a circuit for six variables and six conditions takes roughly three minutes, raising that number to ten variables and conditions can take up to an hour.

By comparison, the time it takes to compute a witness for the compiled circuit is negligible, confirming our initial hypothesis. Our test was conducted much in the same way as the one for compile times. The results are located in the second chart of figure 17.1. For the same arrangement of ten decision variables and ten conditions, we derive a witness for the problem within 90 seconds (as opposed to one hour for circuit compilation). Note also, that the compute time's overall growth curve is much flatter than that of the circuit's compile time. These observations are mostly in line with the theoretical foundations of zkSNARKs, which state that circuit compilation should be much more expensive than witness derivation or even verification.

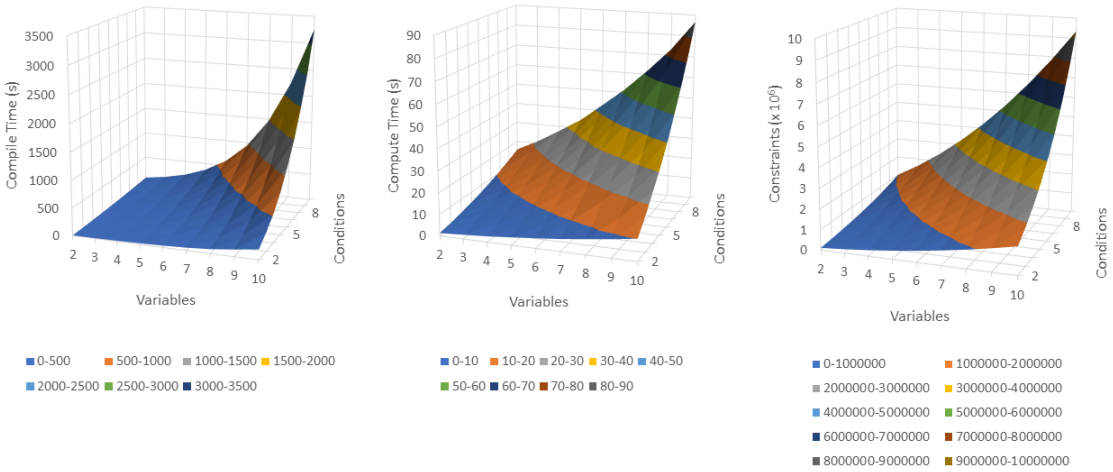


Figure 17.1.: Overview of various performance metrics for different numbers of decision variables and conditions.

Our results are not specific to certain optimizations but hold true for all possible problem instances. Recall from 16.1, that our arithmetic circuit's runtime is fixed to that of a worst-case problem instance, since its size is also fixed. Hence, the test problem we used (with all input values set to zero) runs just as long as any other problem. Therefore we are able to directly compare the results of all of our test runs with each other and conclude the above statement.

Lastly, we tested the total number of R1CS constraints that would be generated by the compiler for a given circuit. While this number is closely related to compile time, they are not the same. Certain operations, e.g. multiplications, produce a relatively low quantity of constraints, but take a significant amount of time to be filled in by the compiler. As a result, the constraint count does not grow at quite the rapid pace as compile time does. This can be seen in the last chart of figure 17.1. More specifically, the number of constraints for a five-by-five optimization circuit is one million compared to ten million for a circuit of ten inputs each. This corresponds to a growth factor less than ten.

If we then also take a look at more practical settings, the obtained results do not support the idea of zkSNARKs as an optimization method, at least not in the near future. This is mainly because real-world optimization problems often require numbers of decision variables and conditions that go beyond anything we were able to show in our analysis. Even the small-scale case study outlined in [4] with 23 participants has one decision variable per household per time step which are themselves subject to twelve different constraints. Compiling circuits for problems of this size is strictly unfeasible, since it would require more work than simply replicating the computation across all nodes.

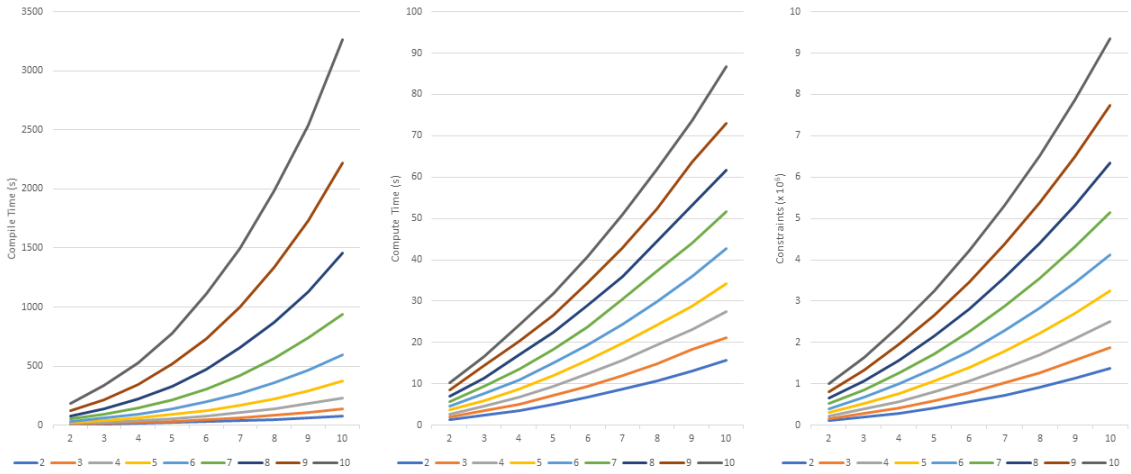


Figure 17.2.: Sensitivity analysis varying the number of decision variables for a fixed number of conditions.

17.2. Sensitivity Analysis

To supplement the results obtained from our initial measurements, we also conduct a sensitivity analysis for both available input parameters. We do this by fixing one of the parameters to a constant value and varying the other, while once again collecting data for the three performance metrics from the previous paragraph. This way, we are able to reveal the impact individual parameters had on the measurements. The results of this can be seen in figures 17.2 and 17.3.

Variables Let us first look at the distribution of compile times with respect to the number of variables. This particular statistic is shown by the first chart in 17.2. First note, that all resulting graphs are polynomials regardless of the chosen number of conditions. When fixing the amount of conditions to lower quantities, however, their rise is barely noticeable, whereas for our maximum condition count, a clear quadratic growth pattern becomes visible. This is in stark contrast to the behavior shown by our total compute times, which is shown in the second chart. Here, an almost linear growth can be observed for all cases. This fact corroborates our previous result and explains why compile time is indeed the main bottleneck of our implementation. Even for the small-scale problems which we have tested in our analysis, the compile time's quadratic growth proves too sever to remain within realistic bounds.

Finally, when we look at the total number of circuit constraints, a similarly counterintuitive pattern as during our initial measurement emerges. Instead of exhibiting the same growth pattern as our compile time measurements, the graphs for total constraint counts, seen in the final chart, very closely resemble those obtained for witness derivation durations. For all ten measured condition levels, increasing the number of decision variables correlates with a quasi-linear increase in constraints. As before, we conclude that this is caused by certain operations in the code which produce a low number of constraints but

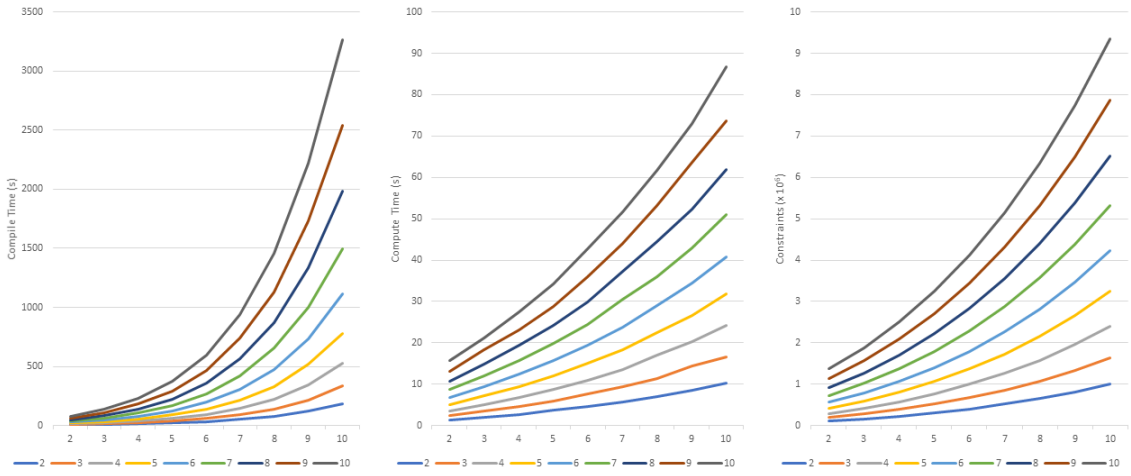


Figure 17.3.: Sensitivity analysis varying the number of conditions for a fixed number of decision variables.

are expensive to compute by the compiler.

Conditions Conversely, let us now conduct a complimentary sensitivity analysis for varying numbers of conditions and a fixed number of decision variables. The results of this analysis can be viewed in figure 17.3. For the most part, we receive similar measurements as before, implying that variables and conditions both contribute to the overall performance of our implementation. Note, however, that the number of conditions seems to have a more significant impact on the total compile time. This can be seen in the first chart, where an exponential growth pattern is clearly visible for larger inputs. This is in line with our intuition that introducing an additional condition to the optimization problem results in a bigger overhead that introducing another variable. On a technical level, this can be explained by the large number of branching circuit paths that are generated for a single condition row of the simplex tableau.

Interestingly enough, the role of our parameters is inverted for the other performance metrics. For instance, the number of decision variables contributes slightly more to the observed witness generation times than the problem's number of conditions, at least within the range defined by our experiment. The same is true for the number of conditions in the compiled circuit. This occurs despite the fact, that the graphs produced by adding extra conditions lose some of their linear shape and start to show a noticeable quadratic curvature, especially for larger inputs, as can be seen in chart two and three of figure 17.3. When pondering why the behave in this way, our hypothesis is twofold: Likely, due to the higher amount of expensive multiplication operations introduced by a new decision variable, the initial overhead is larger for these metrics. Secondly, considering how the growth patterns are slightly different, the scope of our experiment might not capture the full extent of this phenomenon. To fully reveal the impact of individual parameters, further large-scale tests may have to be conducted.

To summarize, it is fair to say the input parameters of our zkSNARK are strongly corre-

lated. The simplex algorithm's runtime scales almost equally with the number of decision variables and conditions. If we increase only one of the inputs, our metrics stay almost level, but a combination of both leads to a rapid decline in performance, especially for circuit compilation. Overall, the performance of our prototype is not yet practical. This can be attributed to multiple factors, but two stand out in particular: On the one hand, zkSNARKs in general are a very young technology and require more work to reach acceptable speeds. On the other, our particular algorithm performs especially poorly on arithmetic circuits due to its high multiplicative complexity. As was the case with the MiMC hash functions, this problem could potentially be solved with completely new optimization algorithms that are better suited for such an environment.

18. Conclusion

In the previous chapters, we have provided a comprehensive overview of the state-of-the-art of blockchain-based verifiable computation. We started out by looking at the three most commonly used techniques in great detail. Trusted blockchain oracles build on existing frameworks and add new trust mechanisms that make them viable for outsourcing program execution. zkSNARKs combine established ideas from zero-knowledge proofs with modern encryption schemes to create a strong cryptographic primitive for verifiable computing. And lastly, MPC adopts many of the same cryptographic protocols as zkSNARKs but applies them to a distributed setting, allowing multiple parties to concurrently calculate a shared secret.

After setting the scene in this way, we followed up with a thorough analysis for the identified technologies. Each technology was evaluated in various categories relating to security, performance and usability. For this we adopted an existing analysis framework and conducted slight changes to better fit our model. By applying this framework to the technologies, we concluded that zkSNARKs are the most promising and future-oriented technology, especially when considering open use cases in the energy sector.

Equipped with this knowledge, we were able to implement a zkSNARK which could solve various energy-related optimization problems. The specific problems for this were chosen based on a use case analysis of the energy sector. The zkSNARK itself was implemented using the ZoKrates circuit compiler and tool chain. By building a working optimizer as a zkSNARK, we were able to show, that energy-related blockchain-based verifiable computing is possible in theory.

Following our implementation, we conducted a performance analysis of the final program. In total, we tested three properties of the circuit: compile time, compute time and number of constraints. The analysis revealed our zkSNARK to be viable for very small problem instances with parameter counts in the single digits. However, as problem size increases, all three examined circuit properties grow unfeasible large, albeit at different rates. This rapid growth quickly leads to zkSNARKs that are not realistic in practical settings. Optimization problems in the real world often have hundreds or thousands of decision variables and conditions, which is clearly unsuitable for our construction.

To conclude, we believe that verifiable computing can be possible in a blockchain environment but still requires more research in the future to become practical. As an area with great potential, VC receives notable interest from the blockchain community with many projects actively being developed. zkSNARKs are the field's current stars but new technologies emerge constantly. With the recent advancements in machine learning, efficient methods for fully homomorphic encryption are now more sough after than ever and may eventually surpass competing approaches. Until then, we might have to rely on untrusted computations just a little longer, unless a big leap in zkSNARK performance is made.

Appendix

A. ZoKrates Source Files

```
include(constants.m4)
from "hashes/mimc7/mimc7R10" import main as hash

from "./simplex" import run as run_simplex


def validate_inputs(bool[INS] sigs_in, field[INS] nums_in, field[INS] dens_in)
    -> bool:

    # ensure no denominator is zero
    for field i in 0..INS do

        assert(dens_in[i] != 0)

    endfor

    return true

#
# Fill in the Simplex table with basic variables.
# Adding the basic variables in code instead of manually inputting them along
    with the non-basic variables
# ensures that the algorithm's table formation laws are respected
# without having to also hash their initial values.
#
def gen_table(bool[INS] sigs_in, field[INS] nums_in, field[INS] dens_in) -> (
    bool[OUTS], field[OUTS], field[OUTS]):

    bool[OUTS] sigs = [false; OUTS]
    field[OUTS] nums = [0; OUTS]
    field[OUTS] dens = [1; OUTS]

    # copy inputs
    for field row in 0..ROWS do
        for field col in 0..COLS_IN do
            sigs[COLS * row + col] = sigs_in[COLS_IN * row + col]
            nums[COLS * row + col] = nums_in[COLS_IN * row + col]
            dens[COLS * row + col] = dens_in[COLS_IN * row + col]
        endfor
    endfor

    # initialize basic variables
    for field row in 1..ROWS do
        for field col in COLS_IN..COLS do
            nums[COLS * row + col] = if row - 1 == col - COLS_IN then 1 else 0
    fi
        endfor
    endfor
```

```
45
46      return sigs, nums, dens
47
48  #
49  # Compute the hashes required for manual input checking.
50  #
51  def compute_hashes(bool[INS] sigs_in, field[INS] nums_in, field[INS] dens_in) ->
        (field[INS]):
52
53      field[INS] hashes = [0; INS]
54
55      for field i in 0..INS do
56
57          field h = hash(if sigs_in[i] then 1 else 0 fi + nums_in[i] + dens_in[i],
        i)
58          hashes[i] = h
59
60      endfor
61
62      return hashes
63
64
65  def main(private bool[INS] sigs_in, private field[INS] nums_in, private field[
        INS] dens_in) -> (field[INS], bool[OUTS], field[OUTS], field[OUTS]):
66
67      bool valid = validate_inputs(sigs_in, nums_in, dens_in)
68
69      field[INS] hashes = compute_hashes(sigs_in, nums_in, dens_in)
70
71      bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens = gen_table(sigs_in,
        nums_in, dens_in)
72
73      sigs, nums, dens = run_simplex(sigs, nums, dens)
74
75      return hashes, sigs, nums, dens
```

Listing A.1: Entry point and pre-processing.

```
1  #
2  # This module implements a static variant of Dantzig's Simplex algorithm.
3  # The algorithm accepts a linear optimization problem in canonical table form
4  # and computes the optimal values for all non-basic variables
5  # as well as the overall result value.
6  # It can be expressed in pseudocode as follows:
7  #
8  # while not optimal and not unbounded:
9  #
10 #   choose variable with largest negative coefficient in objective function as
       pivot column
11 #   if no negative coefficient exists => optimal = true
12 #
13 #   choose condition with smalles positive ratio between total resources
14 #     and resources consumed by the chosen variable as pivot row
15 #   if no positive ratio exists => unbounded = true
16 #
17 #   pivot element := table[pivot_row][pivot_column]
18 #
```

```
#   divide pivot row by pivot element, such that pivot element is 1
#   subtract multiple of pivot row from other rows, such that pivot column becomes 0
#
#


include(constants.m4)
from "./rational" import cmp
from "./rational" import add
from "./rational" import mul


def get_pcol(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens) -> (bool, field):

    # start with first element in objective row
    field pcol = 1
    bool sig_min = sigs[COLS * 0 + 1]
    field num_min = nums[COLS * 0 + 1]
    field den_min = dens[COLS * 0 + 1]

    # iterate over other elements
    for field col in 2..COLS_IN do

        bool sig = sigs[COLS * 0 + col]
        field num = nums[COLS * 0 + col]
        field den = dens[COLS * 0 + col]

        # if new element is smaller
        bool p1 = cmp(sig, num, den, sig_min, num_min, den_min) == 0

        pcol = if p1 then col else pcol fi
        sig_min = if p1 then sig else sig_min fi
        num_min = if p1 then num else num_min fi
        den_min = if p1 then den else den_min fi

    endfor

    # optimal if no negative element exists
    bool opt = cmp(sig_min, num_min, den_min, false, 0, 1) != 0

    return opt, pcol

def get_prow(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens, field pcol) -> (bool, field):

    # start with -1 as the smallest ratio
    field prow = 1
    bool sig_min = true
    field num_min = 1
    field den_min = 1

    # iterate over rows
    for field row in 1..ROWS do

```

```
72          # calculate ratio of resources available vs. resources consumed
73          bool sig_avail = sigs[COLS * row + 0]
74          field num_avail = nums[COLS * row + 0]
75          field den_avail = dens[COLS * row + 0]
76          bool sig_cost = sigs[COLS * row + pcol]
77          field num_cost = nums[COLS * row + pcol]
78          field den_cost = dens[COLS * row + pcol]
79
80          bool sig_cur, field num_cur, field den_cur = mul(sig_avail, num_avail,
    den_avail, sig_cost, den_cost, num_cost)
81
82          # if new ratio is positive and smaller than current
83          bool p2 = cmp(sig_cost, num_cost, den_cost, false, 0, 1) == 2
84          bool p3 = cmp(sig_min, num_min, den_min, false, 0, 1) == 0
85          bool p4 = cmp(sig_cur, num_cur, den_cur, false, 0, 1) == 2
86          bool p5 = cmp(sig_cur, num_cur, den_cur, sig_min, num_min, den_min) == 0
87          bool p6 = p2 && (p3 || p4 && p5)
88
89          prow = if p6 then row else prow fi
90          sig_min = if p6 then sig_cur else sig_min fi
91          num_min = if p6 then num_cur else num_min fi
92          den_min = if p6 then den_cur else num_min fi
93
94      endfor
95
96      # unbounded if no positive value exists
97      bool unb = cmp(sig_min, num_min, den_min, false, 0, 1) == 0
98
99      return unb, prow
100
101 def normalize(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens, field prow,
    field pcol) -> (bool[OUTS], field[OUTS], field[OUTS]):
102
103     bool sig_piv = sigs[COLS * prow + pcol]
104     field num_piv = nums[COLS * prow + pcol]
105     field den_piv = dens[COLS * prow + pcol]
106
107     # iterate over columns and divide by pivot element
108     for field col in 0..COLS do
109
110         bool sig, field num, field den = mul( \
111                 sigs[COLS * prow + col], nums[COLS * prow + col], dens[COLS *
    prow + col], \
112                 sig_piv, den_piv, num_piv \
113         )
114
115         sigs[COLS * prow + col] = sig
116         nums[COLS * prow + col] = num
117         dens[COLS * prow + col] = den
118
119     endfor
120
121     return sigs, nums, dens
122
123 def eliminate(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens, field prow,
    field pcol) -> (bool[OUTS], field[OUTS], field[OUTS]):
```

```
124
125     # iterate over non-pivot rows and subtract multiple of pivot row
126     for field row in 0..ROWS do
127
128         # multiplication factor for current row, such that pivot variable
    becomes 0
129         bool sig_piv = sigs[COLS * row + pcol]
130         field num_piv = nums[COLS * row + pcol]
131         field den_piv = dens[COLS * row + pcol]
132
133         for field col in 0..COLS do
134
135             # pivot row * multiplication factor
136             bool sig, field num, field den = mul( \
137                 sig_piv, num_piv, den_piv, \
138                 sigs[COLS * prow + col], nums[COLS * prow + col], dens[COLS *
    prow + col] \
139             )
140             # subtract from current row
141             sig, num, den = add( \
142                 sigs[COLS * row + col], nums[COLS * row + col], dens[COLS * row
    + col], \
143                 !sig, num, den \
144             )
145
146             sigs[COLS * row + col] = if row != prow then sig else sigs[COLS *
    row + col] fi
147             nums[COLS * row + col] = if row != prow then num else nums[COLS *
    row + col] fi
148             dens[COLS * row + col] = if row != prow then den else dens[COLS *
    row + col] fi
149
150         endfor
151     endfor
152
153     return sigs, nums, dens
154
155 def pivot(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens, field prow, field
     pcol) -> (bool[OUTS], field[OUTS], field[OUTS]):
156
157     sigs, nums, dens = normalize(sigs, nums, dens, prow, pcol)
158
159     sigs, nums, dens = eliminate(sigs, nums, dens, prow, pcol)
160
161     return sigs, nums, dens
162
163 def run(bool[OUTS] sigs, field[OUTS] nums, field[OUTS] dens) -> (bool[OUTS],
    field[OUTS], field[OUTS]):
164
165     bool opt = false
166     bool unb = false
167
168     for field i in 0..VARS do
169
170         opt, field pcol = get_pcol(sigs, nums, dens)
171
```

```
172         unb, field prow = get_prow(sigs, nums, dens, pcol)
173
174         bool[OUTS] sigs_tmp, field[OUTS] nums_tmp, field[OUTS] dens_tmp = pivot(
    sigs, nums, dens, prow, pcol)
175
176         sigs = if opt || unb then sigs else sigs_tmp fi
177         nums = if opt || unb then nums else nums_tmp fi
178         dens = if opt || unb then dens else dens_tmp fi
179
180     endfor
181
182     return sigs, nums, dens
```

Listing A.2: Main algorithmic implementation.

```
1  #
2  # This module is a custom implementation of a rational number type.
3  # A rational number is represented by a tuple (sign: bool, numerator: field,
       denominator: field).
4  # This representation was chosen over other candidates (e.g. floating points),
5  # because of its low multiplicate complexity and the resulting zkSNARK
       performance.
6  # It also prevents some of the rounding errors present in those other techniques
       ,
7  # which can be fatal for the correct execution of the Simplex algorithm.
8  # Addition and multiplication are support out-of-the-box,
9  # subtraction and division can be constructed from addition and multiplication
       respectively.
10 #
11
12
13 from "./signed" import add as add_s
14 from "./signed" import mul as mul_s
15 from "./signed" import cmp as cmp_s
16
17
18 def add(bool sig_l, field num_l, field den_l, bool sig_r, field num_r, field
       den_r) -> (bool, field, field):
19     bool sig_a, field val_a = mul_s(sig_l, num_l, false, den_r)
20     bool sig_b, field val_b = mul_s(sig_r, num_r, false, den_l)
21     bool sig_num, field val_num = add_s(sig_a, val_a, sig_b, val_b)
22     bool sig_den, field val_den = mul_s(false, den_l, false, den_r)
23
24     return sig_num, val_num, val_den
25
26 def mul(bool sig_l, field num_l, field den_l, bool sig_r, field num_r, field
       den_r) -> (bool, field, field):
27     bool sig_num, field val_num = mul_s(sig_l, num_l, sig_r, num_r)
28     bool sig_den, field val_den = mul_s(false, den_l, false, den_r)
29
30     return sig_num, val_num, val_den
31
32 # Compare two rational numbers a and b.
33 # The result is an unsigned number in {0, 1, 2}, where
34 # 0 => a < b,
35 # 1 => a == b,
36 # 2 => a > b.
```

```
#
def cmp(bool sig_l, field num_l, field den_l, bool sig_r, field num_r, field
    den_r) -> field:
    bool sig_a, field val_a = mul_s(sig_l, num_l, false, den_r)
    bool sig_b, field val_b = mul_s(sig_r, num_r, false, den_l)

    return cmp_s(sig_a, val_a, sig_b, val_b)
```

Listing A.3: Operations for rational numbers.

```
#
# This module is a custom implementation of a signed number type.
# A signed number is represented by a tuple (sign: bool, value: field).
# This representation was chosen over other candidates (e.g. two's complement),
# because of its low multiplicate complexity and the resulting zkSNARK
    performance.
# Addition and multiplication are support out-of-the-box,
# subtraction can be constructed from addition, division is never needed.
#


#
# Ensure that the value 0 is always positive.
# This is necessary for the correctness of some operations.
#
def norm_sig(bool sig, field val) -> bool:
    return if val == 0 then false else sig fi

def add(bool sig_l, field val_l, bool sig_r, field val_r) -> (bool, field):

    field val = if sig_l == sig_r then val_l + val_r else if val_l < val_r then
    val_r - val_l else val_l - val_r fi fi
    bool sig = if sig_l != sig_r && val_l < val_r then sig_r else sig_l fi

    sig = norm_sig(sig, val)
    return sig, val

def mul(bool sig_l, field val_l, bool sig_r, field val_r) -> (bool, field):

    field val = val_l * val_r
    bool sig = sig_l != sig_r

    sig = norm_sig(sig, val)
    return sig, val

# Compare two signed numbers a and b.
# The result is an unsigned number in {0, 1, 2}, where
# 0 => a < b,
# 1 => a == b,
# 2 => a > b.
#
def cmp(bool sig_l, field val_l, bool sig_r, field val_r) -> field:
    return \
    if sig_l != sig_r then \
        if sig_l then 0 else 2 fi \
    else \
        if val_l == val_r then 1 \
```

```
46          else if sig_l && val_r < val_l || !sig_l && val_l < val_r then 0 \
47          else 2 fi fi \
48      fi
```

Listing A.4: Operations for signed numbers.

Bibliography

[1] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. Astraea: A Decentralized Blockchain Oracle. *Proceedings - IEEE 2018 International Congress on Cybermatics: 2018 IEEE Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, iThings/Gree*, pages 1145–1152, 2018.

[2] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy Blockchain Oracles: Review, Comparison, and Open Research Challenges. *IEEE Access*, 8:85675–85685, 2020.

[3] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10031 LNCS:191–219, 2016.

[4] Tarek Alskaif and Gijs Van Leeuwen. Decentralized Optimal Power Flow in Distribution Networks Using Blockchain. *SEST 2019 - 2nd International Conference on Smart Energy Systems and Technologies*, pages 1–6, 2019.

[5] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2087–2104, 2017.

[6] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for CPU based attestation and sealing. pages 1–7, 2013.

[7] Merlinda Andoni, Valentin Robu, David Flynn, Simone Abram, Dale Geach, David Jenkins, Peter McCallum, and Andrew Peacock. Blockchain technology in the energy sector: A systematic review of challenges and opportunities, 2019.

[8] Marcos V. M. ARAÚJO, Charles B do PRADO, Luiz F. Rust C. CARMO, Alvaro E. R. Rinc ÓN, and Claudio M. FARIAS. Secure Cloud Processing for Smart Meters Using Intel SGX. *SBSeg*, 18:89–96, 2018.

[9] Ghada Arfaoui, Saïd Gharout, and Jacques Traore. Trusted execution environments: A look under the hood. *Proceedings - 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, MobileCloud 2014*, pages 259–266, 2014.

[10] Li Bai, Mi Hu, Min Liu, and Jingwei Wang. BPIIoT: A Light-Weighted Blockchain-Based Platform for Industrial IoT. *IEEE Access*, 7:58381–58393, 2019.

[11] T. Baroche, F. Moret, and P. Pinson. Prosumer Markets: A Unified Formulation. In *2019 IEEE Milan PowerTech*, pages 1–6. IEEE, jun 2019.

[12] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11959 LNCS:274–302, 2020.

[13] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8642:175–196, 2014.

[14] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[15] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Eprint.Iacr.Org*, (693423):1–83, 2018.

[16] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8043 LNCS(PART 2):90–108, 2013.

[17] Eli Ben-sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Arguments for a von Neumann Architecture. *USENIX Security*, pages 1–35, 2013.

[18] F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on Hyperledger Fabric with secure multiparty computation. *IBM Journal of Research and Development*, 63(2), 2019.

[19] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. *ITCS 2012 - Innovations in Theoretical Computer Science Conference*, pages 326–349, 2012.

[20] Alexander Bogensperger and Andreas Zeiselmair. Updating renewable energy certificate markets via integration of smart meter data, improved time resolution and spatial optimization. *International Conference on the European Energy Market, EEM*, 2020-September:0–4, 2020.

[21] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3), 2014.

[22] Alain Brenzikofer, Arne Meeuw, Sandro Schopfer, Anselma Wörner, and Christian Dürr. QUARTIERSTROM : A DECENTRALIZED LOCAL P2P ENERGY MARKET

PILOT ON A SELF-GOVERNED BLOCKCHAIN 25 th International Conference on Electricity Distribution Madrid , 3-6 June 2019. *25th International Conference on Electricity Distribution*, (June):3–6, 2019.

[23] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK Compilers. pages 677–706, 2020.

[24] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Etherum*, (January):1–36, 2014.

[25] Raymond H. Byrne, Tu A. Nguyen, David A. Copp, Babu R. Chalamala, and Imre Gyuk. Energy Management and Optimization Methods for Grid Energy Storage Systems. *IEEE Access*, 6:13231–13260, 2017.

[26] Fran Casino, Thomas K. Dasaklis, and Constantinos Patsakis. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics*, 36(November 2018):55–81, 2019.

[27] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. *Proceedings - 4th IEEE European Symposium on Security and Privacy, EURO S and P 2019*, pages 185–200, 2019.

[28] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10624 LNCS:409–437, 2017.

[29] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 719–728, 2017.

[30] Tobias Cloosters, Michael Rodler, and Lucas Davi. TEEREX: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. *Proceedings of the 29th USENIX Security Symposium*, pages 841–858, 2020.

[31] Gil Cohen, Ivan Bjerre Damgård, Yuval Ishai, Jonas Kölker, Peter Bro Miltersen, Ran Raz, and Ron D. Rothblum. Efficient multiparty protocols via log-depth threshold formulae (Extended abstract). *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8043 LNCS(PART 2):185–202, 2013.

[32] Mauro Conti, Kumar E. Sandeep, Chhagan Lal, and Sushmita Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys and Tutorials*, 20(4):3416–3452, 2018.

[33] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, pages 1–118, 2016.

[34] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, Nigel Smart, and Peter Scholl. Practical Covertly Secure MPC for Dishonest Majority â or : Breaking the SPDZ Limits Secure Multi-Party Computation Goal : compute. *Esorics*, 8134:1–23, 2013.

[35] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, pages 643–662. 2012.

[36] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 123–140, 2019.

[37] George B. Dantzig, Alex Orden, and Philip Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[38] Dexaran. ERC223. https://github.com/ethereum/EIPs/issues/223, 2017. Accessed: 2020-08-17.

[39] Jacob Eberhardt and Jonathan Heiss. Off-chaining models and approaches to off-chain computations. In *SERIAL 2018 - Proceedings of the 2018 Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 7–12, New York, New York, USA, 2018. ACM Press.

[40] Jacob Eberhardt and Stefan Tai. ZoKrates-Scalable Privacy-Preserving Off-Chain Computations. In *Proceedings - IEEE 2018 International Congress on Cybermatics: 2018 IEEE Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, iThings/Gree*, pages 1084–1091. IEEE, jul 2018.

[41] Alexander Egberts. The Oracle Problem - An Analysis of how Blockchain Oracles Undermine the Advantages of Decentralized Ledger Systems. *SSRN Electronic Journal*, 2019.

[42] Steve Ellis, Ari Juels, and Sergey Nazarov. ChainLink: A Decentralized Oracle Network. 2017(September):1–38, 2017.

[43] Komei Fukuda and Tamás Terlaky. Criss-cross methods: A fresh view on pivot algorithms. *Mathematical Programming, Series B*, 79(1-3):369–395, 1997.

[44] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. Number 1017660, pages 626–645. 2013.

[45] Rosario Gennaro, Stanisław L. Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1592:295–310, 1999.

[46] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*, volume 26, page 169, New York, New York, USA, 2009. ACM Press.

[47] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8042 LNCS(PART 1):75–92, 2013.

[48] GlobalPlatform. TEE Standards. https://globalplatform.org/specs-library/?filter-committee=tee, 2020. Accessed: 2020-08-29.

[49] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, 18(1):186–208, feb 1989.

[50] Jens Groth. On the size of pairing-based non-interactive arguments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9666:305–326, 2016.

[51] J. A. J. Hall and K. I. M. McKinnon. Hyper-Sparsity in the Revised Simplex Method and How to Exploit it. *Computational Optimization and Applications*, 32(3):259–283, dec 2005.

[52] Jonathan Heiss, Jacob Eberhardt, and Stefan Tai. From oracles to trustworthy data on-chaining systems. *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, pages 496–503, 2019.

[53] Jordi Herrera-Joancomartí and Cristina Pérez-Solà. Privacy in bitcoin transactions: New challenges from blockchain scalability solutions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9880 LNAI:26–44, 2016.

[54] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*, volume 108, pages 1–1, New York, New York, USA, 2013. ACM Press.

[55] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[56] Anup Hosangadi, Farzan Fallah, and Ryan Kastner. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2012–2021, 2006.

[57] David G. Humphrey and James R. Wilson. A Revised Simplex Search Procedure for Stochastic Simulation Response Surface Optimization. *INFORMS Journal on Computing*, 12(4):272–283, nov 2000.

[58] Xiaolong Jin, Qiuwei Wu, and Hongjie Jia. Local flexibility markets: Literature review on concepts, models and clearing methods. *Applied Energy*, 261(December 2019):114387, 2020.

[59] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel ® Software Guard Extensions : EPID Provisioning and Attestation Services 1 Introduction 2 TCB Key Binding. *Intel Blogs*, pages 1–10, 2016.

[60] W Kahan. IEEE Standard 754 for Binary Floating-Point Arithmetic. *University of California, Berkeley*, (May 1995):1–30, 1996.

[61] Delaram Kahrobaei, Alexander Wood, and Kayvan Najarian. Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics. *ACM Comput. Surv*, 2020.

[62] M. Karchmer and A. Wigderson. On Span Programs. *Proceedings of the Eighth Annual Structure in Complexity Theory Conference*, pages 102–111, 1993.

[63] Amin Kargarian, Javad Mohammadi, Junyao Guo, Sambuddha Chakrabarti, Masoud Barati, Gabriela Hug, Soummya Kar, and Ross Baldick. Toward Distributed/Decentralized DC Optimal Power Flow Implementation in Future Electric Power Systems. *IEEE Transactions on Smart Grid*, 9(4):2574–2594, 2018.

[64] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing - STOC '84*, volume 4, pages 302–311, New York, New York, USA, 1984. ACM Press.

[65] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10822 LNCS:158–189, 2018.

[66] Balho H. Kim and Ross Baldick. A comparison of distributed optimal power flow algorithms. *IEEE Transactions on Power Systems*, 15(2):599–604, 2000.

[67] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. XJsnark: A Framework for Efficient Verifiable Computation. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:944–961, 2018.

[68] Tsung-Ting Kuo and Lucila Ohno-Machado. ModelChain: Decentralized Privacy-Preserving Healthcare Predictive Modeling Framework on Private Blockchain Networks. 16(4):492–497, feb 2018.

[69] Jae Kwon. TenderMint : Consensus without Mining. *the-Blockchain.Com*, 6:1–10, 2014.

[70] Zhetao Li, Jiawen Kang, Rong Yu, Dongdong Ye, Qingyong Deng, and Yan Zhang. Consortium blockchain for secure energy trading in industrial internet of things. *IEEE Transactions on Industrial Informatics*, 14(8):3690–3700, 2018.

[71] Sin Kuang Lo, Xiwei Xu, Mark Staples, and Lina Yao. Reliability analysis for blockchain oracles. *Computers and Electrical Engineering*, 83:106582, 2020.

[72] Lars Lühr. SGX-hardware list. https://github.com/ayeks/SGX-hardware, 2014. Accessed: 2020-08-22.

[73] Matt Luongo and Corbin Pon. The Keep Network: A Privacy Layer for Public Blockchains. page 11, 2019.

[74] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2429:53–65, 2002.

[75] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. pages 1–1, 2013.

[76] Eric Munsing, Jonathan Mather, and Scott Moura. Blockchains for decentralized optimization of energy resources in microgrid networks. In *1st Annual IEEE Conference on Control Technology and Applications, CCTA 2017*, volume 2017-Janua, pages 2164–2171, 2017.

[77] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

[78] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1592:327–346, 1999.

[79] Charles Noyes. Blockchain Multiparty Computation Markets at Scale. *Ieee Transactions on Xxxxx*, 0(0):1–4.

[80] Kelly Olson, Mic Bowman, James Mitchell, Shawn Amundson, Dan Middleton, and Cian Montgomery. Sawtooth: An Introduction. (January):1–7, 2018.

[81] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Proceedings - IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[82] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 576 LNCS:129–140, 1992.

[83] Xin Pei, Liang Sun, Xuefeng Li, Kaiyan Zheng, and Xiaochuan Wu. Smart Contract Based Multi-Party Computation with Privacy Preserving and Settlement Addressed. *Proceedings of the 2nd World Conference on Smart Trends in Systems, Security and Sustainability, WorldS4 2018*, pages 220–229, 2019.

[84] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. *Whitepaper*, pages 1–47, 2017.

[85] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to Explain Zero-Knowledge Protocols to Your Children. In *Advances in Cryptology â CRYPTO' 89 Proceedings*, pages 628–631. Springer New York, New York, NY, 1998.

[86] Michael Oser Rabin. How To Exchange Secrets with Oblivious Transfer. *Technical Report TR-81, Aiken Computation Lab, Harvard University*, pages 1–5, 1981.

[87] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015*, 1:57–64, 2015.

[88] Srinath Setty. Spartan : Efficient and general-purpose zkSNARKs without trusted setup. *Https://Eprint.Iacr.Org/2019/550*, pages 1–25, 2019.

[89] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, nov 1979.

[90] Shantanu Sharma and Wee Keong Ng. Scalable, On-Demand Secure Multiparty Computation for Privacy-Aware Blockchains. In *Communications in Computer and Information Science*, volume 1156 CCIS, pages 196–211, 2020.

[91] Tiago Sousa, Tiago Soares, Pierre Pinson, Fabio Moret, Thomas Baroche, and Etienne Sorin. Peer-to-peer and community-based markets: A comprehensive review. *Renewable and Sustainable Energy Reviews*, 104:367–378, 2019.

[92] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. pages 1–50, 2019.

[93] Fabian Vogelsteller and Vitalik Buterin. ERC20. https://eips.ethereum.org/EIPS/eip-20, 2015. Accessed: 2020-08-17.

[94] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. *Proceedings - IEEE Symposium on Security and Privacy*, 2018-May:926–943, 2018.

[95] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, Xiao Feng Wang, Vincentchenguo Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *Proceedings of the ACM Conference on Computer and Communications Security*, (May):2421–2434, 2017.

[96] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in intel SGX enclaves. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9878 LNCS:440–457, 2016.

[97] Jiani Wu and Nguyen Khoi Tran. Application of blockchain technology in sustainable energy systems: An overview. *Sustainability (Switzerland)*, 10(9):1–22, 2018.

[98] Andrew C. Yao. Protocols for Secure Computations. *Annual Symposium on Foundations of Computer Science - Proceedings*, pages 160–164, 1982.

[99] Randy Yates. Fixed-Point Arithmetic : An Introduction. 2001.

[100] Jose M. Yusta, Gabriel J. Correa, and Roberto Lacal-Arántegui. Methodologies and applications for critical infrastructure protection: State-of-the-art. *Energy Policy*, 39(10):6100–6119, 2011.

[101] Andreas Zeiselmair and Alexander Bogensperger. Development of a System Cartography and Evaluation Framework for Complex Energy Blockchain Architectures (in review). In *Internationaler ETG-Kongress 2021*, Wuppertal, 2021. VDE ETG.

[102] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. *Proceedings of the ACM Conference on Computer and Communications Security*, 24-28-Octo:270–282, 2016.

[103] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876. IEEE, may 2020.

[104] Jinquan Zhao, Zhenwei Zhang, Jianguo Yao, Shengchun Yang, and Ke Wang. A distributed optimal reactive power flow for global transmission and distribution network. *International Journal of Electrical Power and Energy Systems*, 104(June 2018):524–536, 2019.

[105] Hanrui Zhong, Yingpeng Sang, Yongchun Zhang, and Zhicheng Xi. Secure Multi-Party Computation on Blockchain: An Overview. pages 452–460. 2020.

[106] Guy Zyskind. Efficient Secure Computation Enabled by Blockchain Technology. 2016.

[107] Guy Zyskind. Enigma: Decentralized Computation Platform with Guaranteed Privacy. *New Solutions for Cybersecurity*, pages 1–14, 2019.