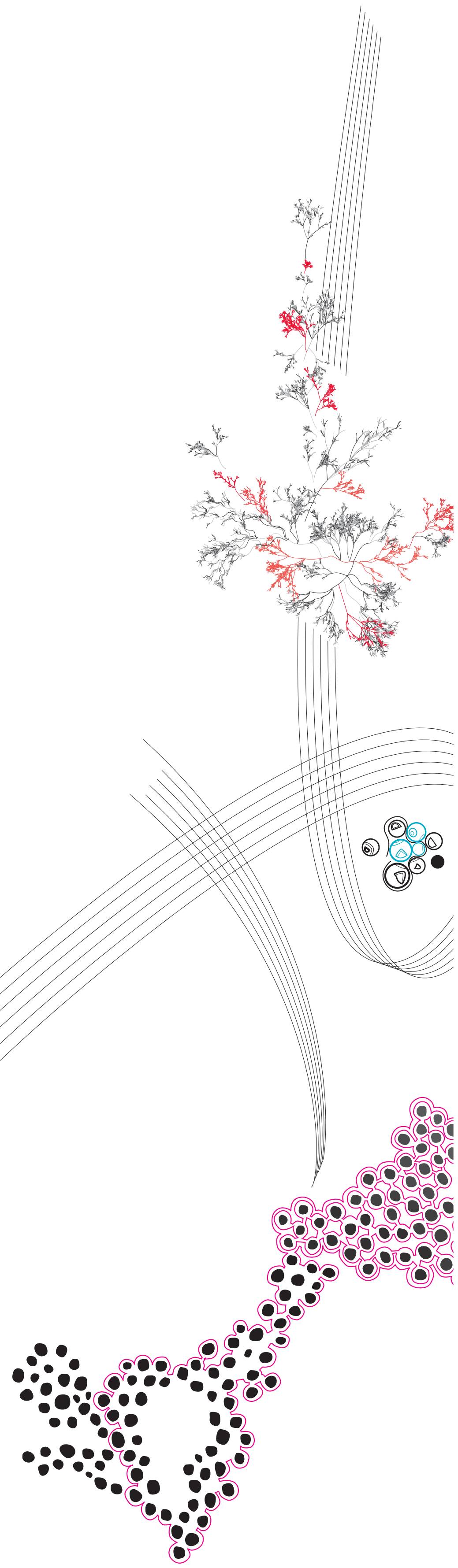
MSc Computer Science
Final Project



Benchmarking the zk-SNARK, zk-STARK, and Bulletproof Non-Interactive Zero-Knowledge Proof Protocols in an Equivalent Practical Application

Bjorn Oude Roelink

Supervisor: Mohammed El-Hajj

June, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

This research work constitutes a follow-up research to our previous Systematic Literature Research (SLR) [95], which examined the applications and performance of the zk-SNARK, zk-STARK, and Bulletproof Non-Interactive Zero-Knowledge Proof (NIZKP) protocols across a variety of collected works. This work designed and implemented a benchmark comparing the same three NIZKP protocols using an equivalent real-world application to fill one of the observed research gaps. By realizing and benchmarking a dynamic MiMC hash application using four general-purpose programming libraries across two programming languages, we could compare the performance between each of the three protocols and conclude the application contexts best suited for each. Our results showed that the zk-SNARK protocol produced the smallest proofs, whereas the zk-STARK proofs were the largest overall. Regarding the proof generation and verification times, we noticed the zk-STARK protocol to be the fastest on average while the Bulletproof protocol was the slowest in both metrics. These observations proved to be in line with the general notion of the protocol performance ordering found online, except that the zk-SNARK proofs verified marginally faster than zk-STARK proofs in our benchmark as opposed to the contrary being reported elsewhere. We established that this work constitutes a contribution to the scientific knowledge on the functionality, security, and especially the performance aspects of each of the three main NIZKP protocols and that it furthermore facilitates the usage of NIZKPs in practical applications by equipping interested parties with sufficient knowledge to make informed decisions on which of the three protocols to use.

Keywords: non-interactive zero-knowledge proof, zk-SNARK, zk-STARK, Bulletproofs, privacy-preserving, authentication, application, benchmark, performance, security

Chapter 1

Introduction

In this chapter, we start off with a description of the background and corresponding context for this work in section 1.1. The section also describes the research that we intend to perform in this work, in addition to why it is relevant. We then define the research questions for this work, as well as the aims and objectives we set out to achieve, in section 1.2 and section 1.3 respectively. We subsequently describe the scope of this work, including scope limitations, in section 1.4, and elaborate on the relevance of this work in section 1.5. Finally, we describe the organization of the remainder of this work in section 1.6.

1.1 Background & context

In life, it is a regular occurrence that individuals want to prove a statement to another person. The simplest, most direct, and arguably the most conducted ways for one to do so is by plainly stating, explaining, or showing the answer to the statement in a manner that the other person can verify themselves. For example, if a customer wants to purchase some age-restricted goods, then they state that their age meets or exceeds the minimum age requirement. They can prove this by showing the cashier some form of identity document, who can then verify the document to be valid and that the birth date indeed shows the customer to be above the required minimum age. While this process is sufficient for the verifier, it may not always be desirable for the prover since it can expose more information than is required to prove the statement. In our example, the cashier cannot only verify that the customer is above the minimum age requirement, they also obtain knowledge on the exact birth date and other identifying information of the customer. In addition, this is an even more substantial problem in digital environments. Whereas in real life the customer can verify that they received their identity document back and that no copy was made in the process, we cannot say the same for the digital environment where any server could have stored a copy of the identity information. This constitutes just a simple example, other examples to think of include: How can a landlord verify that a potential tenant meets a minimum income requirement without obtaining their exact income figures? How can a customer verify that their cloud provider has truthfully performed a computation on the provided input data, without said customer performing the entire computation again themselves locally?

Zero-Knowledge Proofs (ZKPs), first introduced in a work by Goldwasser et al. [59], are a recent technology that could solve these problems. ZKPs allow a prover to prove a given statement, the proof of which a verifier can subsequently verify without being able to obtain any knowledge besides the facts induced by the correctness of the statement itself.

However, traditional ZKPs are interactive, meaning that they require multiple interactions between the prover and verifier before the verifier can trust or reject the statement. In addition, other parties cannot verify the same proof afterward since this would require additional interactions. This limits the practicality of standard ZKPs. To this end, Blum et al. proposed Non-Interactive Zero-Knowledge Proofs (NIZKPs) [22]. NIZKPs enable a verifier to verify a claim in a single interaction, while also allowing other verifiers to verify the truth of the proven statement at another point in time.

Notably, ZKPs, especially the non-interactive variants, have gained prominence in cryptocurrencies like ZCash [122] and Ethereum [123]. In these contexts, they facilitate transaction verification without disclosing sensitive transaction details, thereby preserving privacy. Although cryptocurrencies have been the main source of interest in ZKPs, due to their surge in popularity next to other blockchain technologies, the utility of ZKPs extends far beyond this domain.

In our previous Systematic Literature Review (SLR) work [95], a summary of which we detail in chapter 2, we collected applications of the three main NIZKP protocols relating to privacy-preserving authentication. Notably, we investigated applications and the performance of the zk-SNARK (zero-knowledge Succinct Non-Interactive Argument of Knowledge) [97] [63], zk-STARK (zero-knowledge Succinct Transparent Argument of Knowledge) [14], and Bulletproof [33] protocols. In the SLR work, we examined a total of 41 works that applied NIZKP protocols in a diverse set of applications. However, we found high variability in protocol performance metrics between the several applications, which we believed to be attributable in large part to the difference in applications and benchmarking procedures. This result indicated that a research gap exists for a comparison of the three main NIZKP protocols benchmarked in an equal, real-world applicable, use case.

Our aim in this work is to satisfy the observed research gap by performing a benchmark of the three main NIZKP protocols implemented in an equal, real-world privacy-preserving related, application. The relevance of this, which we further detail in section 1.5, lies mostly with researchers and application designers obtaining a meaningful overview of the main NIZKP protocols, the situations in which they excel, and their implied performance characteristics. Insights from this work can furthermore guide researchers to the main aspects of concern when applying NIZKP protocols to real-world applications. This, in turn, can incite research into mathematical improvements and newly designed NIZKP protocols that reduce the deficiencies of existing protocols.

1.2 Research questions

To define our aims and objectives for this research in section 1.3, we first outline the key research questions that we intend to address as a result of this research work. These questions serve to guide the main direction of this research investigating the differences between the zk-SNARK, zk-STARK, and Bulletproof protocols:

1. What are the performance differences between the three included NIZKP protocols, as observed from a real-world implementation of each protocol in an application that is as equal as possible, expressed in efficiency and security level?

2. What use case contexts are most beneficial for each NIZKP protocol, given the unique combination of its features and performance metrics?

In our previous SLR work [95], the applications described in the included research works were each implemented with a single protocol. This meant that the research works were hard to compare on common grounds because of the dissimilar applications, benchmark procedures, and results. The objective of this research is therefore to implement a single application for the three protocols in a manner that is as similar as possible, with the direct purpose of making comparisons between the three protocols more straightforward. As a result, the comparison outcomes should be more informative. This objective is deeply embedded in the previously stated research questions, meaning that these questions will guide us towards a deep exploration of the three NIZKP protocols in a manner that aims to expose and clarify their associated differences.

1.3 Aims & objectives

We now reflect on the aims we set for our overall research, specifying the aims that we were unable to fulfill to our expectation in the SLR. These aims were to fill the research gap in comparing the three most used NIZKP protocols and to provide recommendations on the settings in which each protocol is most advantageous. The objectives we therefore set to achieve in this research work were:

1. Create an implementation and evaluate the protocols in a practical setting, using a common benchmark for a real-world use case.
2. Create a comparison of the efficiency and security of these three protocols, including their trade-offs between efficiency and security.
3. Describe recommendations for the use of these protocols in different applications, based on their strengths and weaknesses.

While we made advances on these objectives in our previous SLR work, we intend to further progress in the development of understanding related to these aims. This specific research work therefore aims to more comprehensively achieve the stated objectives to determine conclusive answers to the research questions from section 1.2.

To conclude, our aims and objectives for this research are to further detail the performance characteristics of the three most prevalent NIZKP protocols. We aim to do so by more comprehensively comparing those protocols in a benchmark, where we implemented each protocol in an application that is as equal as possible between the three implementations. We can then thoroughly answer which aspects of each NIZKP protocol should be considered when choosing a protocol to be applied in a particular environment.

1.4 Scope & limitations

The scope of our research is twofold.

First, we succinctly describe the mathematical and cryptographic primitives underlying each of the three main NIZKP protocols in chapter 4, the intention of which is to provide a concise understanding of the fundamental techniques that differentiate them. We do not, however, aim to accomplish a comprehensive mathematical and cryptographic manual that can be used as the basis for implementing the protocol itself in code or to create a new protocol from scratch. Furthermore, in the same chapter, we describe the security model of each protocol, next to some vulnerabilities that have surfaced in at least some of the NIZKPs included in this work. The intention for these is, again, not to be comprehensive, instead, the information should serve as a general overview of security aspects and security vulnerabilities to consider when choosing a NIZKP protocol.

Second, this work designs and performs a benchmark comparing the three NIZKP protocols zk-SNARK, zk-STARK, and Bulletproofs on their performance and security level. In the benchmark, each protocol implements an as equal as possible, privacy-preserving authentication-related, application using general-purpose programming libraries that implement each protocol. There are several limitations to this part of our scope. First, we intend to implement each protocol in an application to enable straightforwardly comparing their performance. For this, the application should be as equal as possible. The application, however, does not have to consider and implement each aspect that a production-ready real-world application would, as long as the benchmark results are representative. Second, we implement each protocol in a single application. We do not implement multiple application benchmarks and will not implement the benchmark application for an exhaustive selection of programming languages and NIZKP protocol libraries. Provided that our benchmark implements the application using at least each of the NIZKP protocols, we realized this scope. Finally, while we aspire to benchmark the security level of each protocol, we will not designate time for an in-depth attempt at breaking the security for each protocol. We leave this up to other researchers, as this is more meaningful to perform in the context of an actual production-ready application anyway.

1.5 Relevance

As hinted at in section 1.1, the relevance of this work is situated in the information it provides for researchers and application designers who consider using a NIZKP to provide privacy in an application.

The overview of primitives and security facets of each NIZKP protocol, though mainly intended to provide context on the origin of security and performance differences, presents an excellent overview of the aspects and features that each protocol is comprised of. While researchers could obtain identical information by combining a variety of sources, as we did to obtain the knowledge included in this work, this would constitute an abundance of work that we argue they could better spend on other facets of their research or application. Besides, the cited sources present an excellent subsequent reading for anyone for which the included information is unsatisfactory.

The main relevance of our work, however, arises from information derived from the benchmark results. By carefully inspecting, analysing, and discussing the results of the benchmark, we can compare the three protocols in a novel way never realized before. While, as we further discuss in section 7.1, there is some information on the Internet comparing the performance of the same three NIZKP protocols [120] [90] [93], it was unclear to us where these metrics originate and how the benchmark was performed. To the best of our knowledge, our research constitutes the first work that extensively described the performed benchmark from a scientific perspective, thereby it is repeatable and allows for the extraction of a plethora of knowledge. This constitutes a contribution to the scientific knowledge of NIZKP protocols by providing a dependable source for information on the performance differences distinguishing the main NIZKP protocols.

We anticipate our work to mainly benefit two groups of people. The first group is a population of researchers new to the concept of NIZKPs who intend to obtain knowledge on the main protocols. For this community, our work constitutes a welcome addition to the SLR we previously performed [95] summarized in chapter 2. The second group contains anyone who intends to apply either of the three NIZKP protocols to an application, and who want to know the distinct characteristics and performance aspects to consider when choosing between the protocols. This could benefit e.g. academic research on a novel way to provide privacy, trust, or improved performance in an application. It could also benefit a business project, in which a corporation intends to bring a novel concept that uses NIZKP protocols to market. It could even benefit society in general, by enabling individuals to acquire the required knowledge on the technology of NIZKPs. They could then use this to successfully choose and apply a NIZKP protocol in a personal project, which they could publish under an open-source license for the benefit of less technical members of society.

Altogether, we believe that our work has the potential to directly benefit certain groups of researchers and application designers, eventually indirectly benefiting every member of society.

1.6 Organization

The organization of the remainder of this work is as follows. We summarize our previous systematic literature research work in chapter 2. This summary includes our findings and the outcomes and additionally explains why we proposed this follow-up research. With the reasoning for this research made clear, we continue in chapter 3 by thoroughly describing the methodology that we applied to our investigation on NIZKP protocols. We depict the approach and design for the NIZKP protocol benchmark, next to the analysis that we sought to perform on the results. Before implementing our benchmark, though, we first digress slightly in chapter 4, which describes the mathematical and cryptographic primitives behind each of the three protocols. We will explain the importance of this chapter in its first section. With a clear understanding of what differentiates each NIZKP protocol, we continue in chapter 5 by describing the exact setup we used to implement the benchmark. This chapter lists the software and hardware that we used in the implementation and reports the details of the exact benchmark implementation and subsequent benchmarking procedure. Performing this benchmarking procedure should provide us with all required benchmark results, which we detail and analyse in chapter 6. In chapter 7 we then discuss the results obtained from the benchmark. This discussion leads us to mention the results we achieved, after which we answer the research questions we had for this work. Furthermore, we discuss the strengths, limitations, and significance of our work, wrapping up the chapter with a section on the potential applications for the findings in our work. Finally, we conclude this work with some closing remarks in chapter 8. Included in this chapter are the main findings of this work, some recommendations on the use of NIZKP protocols, and some potential future research directions.

Chapter 2

Literature review

In this chapter, we summarize our previous SLR work [95]. To start, we first summarize the SLR and detail our findings in section 2.1, additionally highlighting the observed key trends. From the observation and key trends, we remark on some research gaps in the current literature. We succinctly describe these research gaps, and the limitations of our SLR, in section 2.2. With the key trends and observed research gaps out of the way, section 2.3 closes this chapter by describing how we aim to address one of the observed gaps and our SLR limitations in this research, expanding on previous research to increase scientific knowledge. There, we also include our rationale for developing this work in the first place.

2.1 Summary & findings

In our previous SLR work, we analyzed a broad spectrum of research works that described diverse use cases related to authentication. All included works were related because of our requirement that the use case applied at least one of the three NIZKP protocols, zk-SNARK, zk-STARK, or Bulletproofs, for some privacy-preserving use within the application context. Ultimately, we examined 41 research works that surfaced from our collection and filtering criteria, discussing their implementation of the NIZKP protocol, and comparing these implementations on their use case. Furthermore, we discussed the performance and security of the NIZKP in the application when a work included benchmarked figures for these. For anyone interested in a more detailed description of our SLR intentions, collection and filtering process, results, and discussion, amongst other things, we recommend consulting the full research document [95]. We limit the remainder of this section to highlight the key findings from the SLR.

To start, 31 of the 41 works included in our SLR employed the zk-SNARK protocol in their described application, whereas the other 10 works utilized the Bulletproof protocol. This indeed means that our work did not end up including any works that based their application on the zk-STARK protocol. While this prevented us from drawing definitive conclusions on the proportionate use of the zk-STARK protocol compared to the other protocol, we did remark that this finding signifies the zk-STARK protocol was not commonly deployed in privacy-preserving authentication-related applications. More specifically, applications adhering to the search and filtering criteria from the SLR do not seem to utilize the zk-STARK protocol. We exert confidence in the notion that the reason for this will be more evident by the end of this work.

We also want to recite the observation that all but two works did not mention the quantum resistance of their implementation. We find this interesting especially since none of the 41 included works applied the only quantum resistant protocol, zk-STARK. This clearly emphasizes a lack of consideration regarding this security aspect, despite quantum computing and quantum-resistant cryptographic protocols having been an ongoing important topic for the past few years [37].

Of the 41 works included in the SLR, 30 works included some form of performance analysis of the implementation. Among those, 22 employed the zk-SNARK protocol, with the remaining eight works utilizing Bulletproofs. In the SLR we discussed the performance results in several categories, though here we will only review the overall performance differences between all works. We observed highly varying measures in multiple categories of performance metrics, including the proof size, proof generation time, and proof verification times. These variations were significant, with several orders of magnitude performance difference between the same protocol applied in different works. Considering this extreme variance in observed metrics, we concluded that it was impossible to draw any definitive conclusions from comparing the performance between applications. The research works would have to specifically perform their benchmarks in a related way to another research work for us to draw any revealing conclusions from the comparison.

We had to draw a similar conclusion to that of the performance comparison for the security comparison, which proved to be even more complex to perform and accomplish a reasonable comparison from. The main reason for this difficulty was the diverse ways researchers used to describe the security of each implementation. Some works described the security by proving mathematical theorems in either natural language or as mathematical statements, whereas others described the security requirements of their application and mentioned either how they were achieved or how attacks were mitigated through implemented security measures, just to name a few of the encountered possibilities. Altogether, our SLR work had a particularly challenging time inferring any reliable security comparison outcomes from the 31 works that included some form of security analysis.

2.2 Research gaps

To remediate the current impossibilities of comparing different applications and their applied protocols on their performance and security, as described in section 2.1, we suggested future research into a benchmarking standard. More concretely, we stated that the following actionable question arose from our SLR: *"How can future security analyses of non-interactive zero-knowledge proof application implementations be standardized to facilitate better comparison?"* When every research work utilizing NIZKP protocols would follow such standard, it would facilitate a more uniform benchmarking procedure which enables an equitable and in-depth performance comparison between works. Yet, as our SLR found multiple research gaps stemming from limitations in current research works, this is not the research direction that we took for this work.

The research gap that we intend to address in this work is the lack of availability, to the best of our knowledge, of a comprehensive applied performance comparison on the three main NIZKP protocols. Such benchmarks should utilize each of the zk-SNARK, zk-STARK, and Bulletproof protocols in an identical application to allow anyone to extract meaningful metrics from the benchmark. In the next section, we explain how we will approach to addressing this research gap.

2.3 Addressing research gaps

This work intends to perform the benchmark described in section 2.2 to fill the previously stated research gap. This means that we will describe, in detail, the design and implementation of a benchmark application that we implemented as equally as possible for each of the three NIZKP protocols. To achieve such implementation, we select at least one programming library for each of the zk-SNARK, zk-STARK, and Bulletproof protocols, and use these libraries to implement an identical application design. We can then conduct the benchmarking procedure, which we meticulously define in this document, and thereby obtain metrics on the performance of each protocol implementation. This data we then use to compare the protocols on their performance facets, to conclude, and to provide recommendations on which situations warrant the usage of each protocol given their features, performance, and security characteristics.

The design of our benchmark will inherently incur some limitations on the results that we obtain, in turn limiting the indications we can provide from a comparison using these metrics. For the exact cataloged limitations please refer to section 1.4. We, however, express our conviction that the benchmark results will be beneficial for improving scientific knowledge on the NIZKP protocols regardless of the limitations and that the comparison will furthermore help many researchers obtain knowledge on the performance and security aspects embedded in each protocol.

Overall, we considered the stated knowledge gap to be important to fill given the rise in popularity of NIZKPs which we previously observed in our SLR from the increasing number of published research works by year utilizing NIZKP protocols (see Figure 5 in our SLR [95]). Being well-informed on the performance and security characteristics of each protocol is an important first aspect of selecting the right protocol for a given application. A comparison between the three main NIZKP protocols implemented in an identical application, as proposed by this work, could therefore strengthen the current corpus of scientific knowledge on this topic.

Chapter 3

Methodology

In this chapter, we detail the methodology that we applied to obtain an answer to the research questions. We define an approach in which we describe how we aimed to achieve the defined objective in section 3.1. Then, in section 3.2, we describe in a detailed manner the design of our benchmark, as well as the application on which we benchmark the three NIZKP protocols. Finally, we outline the results that we intend to obtain from the benchmark and the analyses that we will conduct on the acquired data in section 3.3 and provide a schematic overview of our work in section 3.4.

3.1 Approach

As we previously stated, the main approach of this research was to design a benchmark that implements the same application, or as close as possible, for each of the NIZKP protocols. For this, we used general-purpose programming libraries that implement the three types of NIZKPs of interest: zk-SNARK, zk-STARK, and Bulletproofs. This would give us the ability to directly compare the metrics collected from the benchmark between the protocols, or at minimum the metrics available for all three. The benchmark should preferably use a full-featured, stable programming library to implement the NIZKP application since this provided us with the most options, stable performance, and a hopefully somewhat optimized codebase. Additionally, we preferred for all three protocol libraries to use the same programming language, since this would remove the variable of different performance and options of different programming languages. We also expressed a preference for low-level compiled languages over higher-level interpreted languages, to reduce runtime overhead and performance variability. We required the NIZKP libraries to be intended for general-purpose use, meaning that they were usable for all kinds of proofs in various application settings. While it would have technically been possible to implement a custom NIZKP protocol implementation for one specific application, enabling optimisations for that specific application, we wanted our benchmark to be representative of all kinds of different applications. Furthermore, while we only implemented a single application in our benchmark, by using general-purpose NIZKP libraries for each protocol the performance differences between the protocols can be generalized for many other applications. We implemented the benchmark in code using the same programming language that the NIZKP libraries were written in, which enabled us to perform benchmarks directly on individual parts of the code. This was a requirement for us because we needed to benchmark the separate phases of the protocol, namely the setup, proving, and verification phases. Implementing the benchmark in this manner furthermore allowed us to access the size and security level metrics provided by the programming languages and NIZKP libraries. Both metrics would have been harder to benchmark accurately when running a benchmark using just compiled binaries as input.

3.2 Design

As described in our approach, we aimed to design an application, preferably a privacy-preserving authentication-related one, that was as equal as possible between the three NIZKP protocols. This enabled us to conduct a benchmark to obtain our benchmark results on the performance differences between the three NIZKP protocols. To implement such a benchmark, we first had to conceive an application that was possible to implement for all three NIZKP protocols.

Our initial design idea was inspired by a Cloudflare blog post on replacing CAPTCHAs with attestation of personhood using Hardware Security Keys (HSKs) [83]. Whalen et al. [119] elaborated on that blog post in their research. To summarize, the main idea was that, instead of performing a CAPTCHA, the user must provide a signature that validates with one of the trusted HSKs signature keys. If the signature validates using one of the trusted keys, then an HSK from a trusted manufacturer must have signed the message. Because the initiator trusts that manufacturer, they can subsequently trust that the HSK requires a user to touch the security physical device to sign the message. This in turn attests to the personhood of the user and removes the requirement for the user to prove that they are not an automated process by filling out a CAPTCHA. At the end of their work, the authors described a potential future work to provide a privacy enhancement to the current idea, since the current attestation implementation leaked a hard-coded certificate associated with the HSK. While this certificate was not unique and shared across a batch of at least 100000 devices, as per the FIDO UAF protocol specification section 4.1.2.1.1 [78], preventing the attestation from leaking the used device certificate would further strengthen the privacy aspect by preventing identification of groups of users.

This work spawned a follow-up work by Faz-Hernández et al. [50], with a corresponding blog post [75]. This follow-up work described how the authors achieved attestation of personhood using hardware security keys without leaking the device certificate details, which they called zkAttest. In summary, zkAttest used sigma-protocol ZKPs on a committed public key to verify that the key was used to generate either an Elliptic Curve Digital Signature Algorithm (ECDSA) signature or a Schnorr signature. The ZKP ensured the used public key was part of a list of public keys from trusted HSK while keeping the exact used key private, using the idea of ring signature schemes for the attestation. By verifying the proof, the verifier could ensure that they trust the HSK used by the prover to generate the signature, without learning anything about the used public key aside from them trusting it.

The zkAttest work, as stated, used a specialized sigma-protocol implementation for their NIZKP. While the paper mentioned that zk-SNARKs would also work, they chose the specialized sigma-protocol implementation for several reasons. These reasons include that designing and compiling a SNARK was "tricky", that there were few SNARK tools targeting JavaScript, that the SNARK required unfalsifiable assumptions even under the CRS model, and that the CRS would have been large. For our benchmark, however, these reasons were less of an obstruction and could even show the limitations of the zk-SNARK protocol. Despite that, for reasons which we will more comprehensively describe in chapter 5 section 5.3, we were unable to implement this application within the confounds of our research. In brief, we were limited by having to implement the same application for the three NIZKP protocols, the requirements for the NIZKP libraries, the building blocks provided by each library, and the time allotted for our research.

Given these limitations, which meant we could not implement the attestation of personhood in zero-knowledge idea, we resorted to a much simpler alternative. Namely, we decided to implement a specific hash function in each of the three protocols as the application for the benchmark. Even this implementation has some limitations, which we will elaborate on in section 5.3. However, we found that this idea would provide the most equal application between the three protocols that we could implement within the time allotted to our research. The main problem would be the limiting constraints of each protocol, as observed when exploring options to implement the attestation of personhood idea.

Even though our idea to implement a hash function as the benchmark application does not strictly classify as a privacy-preserving authentication application, we justify our decision by stating that hash functions are an important building block for many more complex privacy-preserving authentication applications. This means that even though we do not directly benchmark such a more complex application, the resulting benchmark should nonetheless be indicative of the typical performance to expect when implementing an elaborate privacy-preserving authentication application using one of the benchmarked NIZKP protocols. Furthermore, for reasons we will elaborate on in section 5.3, we could not produce a more intricate application that would provide with as equal a benchmark between the three protocols as the simple hash function. In addition, the hash function idea allowed us to, depending on the hash function, increase the number of rounds or chain the hash to increase the number of computations performed in the benchmark. This enabled us to benchmark the three protocols with more or less work, meaning that we gained insight into how the performance of each ZKP protocol scaled. This insight was of essence for our work given that it indicates what performance one could obtain from each protocol in a range from simple calculations to intricate proof circuits.

3.3 Results analysis

Now that we defined our approach for the benchmark, we conclude the methodology by indicating which metrics we aimed to collect from the benchmark and defining the analyses that we intended to conduct on those metrics.

For the analysis, one should note that the obtainable metrics differed between the protocols. For example, the zk-SNARK protocol requires a trusted setup, while the zk-STARK and Bulletproof protocols do not. This meant that for the zk-STARK protocol, we were interested in the size of the CRS, while this metric was not available for the other two protocols. The metrics that we intended to acquire for all three protocols included the proof size, the proof generation time, and the proof verification time. We also strove to obtain the conjured and proven security levels of the proofs. As we clarify in section 5.3, however, we could not collect this information for each protocol identically. Finally, there were some metrics for which the availability depended on how each library implements the ZKP protocol. For example, when the library required additional compilations, or when it did not include commitments in the proof. We strove to provide all metrics relevant to each protocol, facilitating a proper comparison between the protocols on the aspects of data transfer, storage size, and computation times.

As for the analyses, we evaluated at least the following aspects of the protocols:

- Setup requirements and time: What requirements were there for trusted setup in each protocol? How long did this setup take and what was the size of the data that had to be shared?

- Proof generation: How long did generating the proof take? What was the size of the data resulting from the proof required for verification of the proof?
- Verification: How long did it take to verify the proof?
- Security aspects: How did the security level differ between the protocols? And where applicable, how did an increase or decrease in the security level impact the other metrics?

Furthermore, we will also comment on aspects of the protocols and their respective library implementations that cannot be expressed in exact metrics. Most importantly, whether we observed aspects of a particular implementation making a library or protocol especially useful or entirely impractical in certain applications, given their situational demands.

3.4 Overview

To conclude this chapter, we provide a schematic overview of the entire process for our research work, including the previously performed SLR, in Figure 3.1.

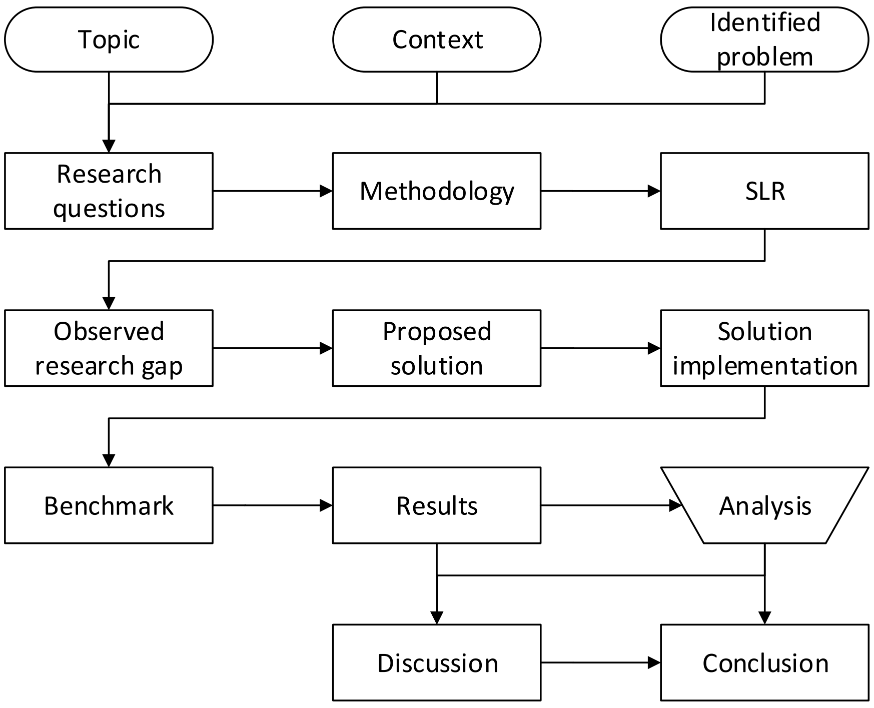


FIGURE 3.1: Schematic overview of our research work.

Chapter 4

Primitives

In this chapter, we describe the primitives underlying the three NIZKP protocols. The main goal of this chapter is to have a clear understanding of the details of the protocols that we will use in our benchmark. First, we describe why it is important to have a limited level of knowledge on the primitives behind the three NIZKP protocols described in this work in section 4.1. We subsequently provide a concise overview of the mathematical primitives underlying the zk-SNARK, zk-STARK, and Bulletproof protocols in section 4.2, section 4.3, and section 4.4 respectively. We then describe the security models and assumptions for each protocol in section 4.5 and continue with descriptions and examples of known vulnerabilities in section 4.6. To conclude the chapter, we briefly restate the defining characteristics of each protocol in a summarizing comparison in section 4.7.

4.1 Importance

To start off this chapter, we first state the reason for including this chapter: to understand where performance and security differences between the different protocols originate. The mathematical and cryptographic primitives underlying a NIZKP protocol are not only the source for its functionality, proving a statement in a succinct and privacy-preserving manner, but they furthermore establish their primary features, strengths, and limitations, which in turn are reflected in the performance and security characteristics. So, to understand the full picture of this work, including the source of the performance and security differences that we aim to benchmark and the conclusions that follow, it is important to know how the protocols differ underneath. Being aware of the differences underlying the performance and security of a protocol enables some level of intrinsic understanding of how protocols and their features work, allowing one to evaluate when a given protocol is viable to utilize in their use case in the first place.

Besides the performance primitives behind each protocol, this chapter also describes the security models and assumptions each protocol makes. These security models and assumptions are essential knowledge for anyone applying NIZKPs to their use case since the protocols will not provide the security that may be expected in situations when an implementation violates these models and assumptions. Alarming situations can in those circumstances result when the NIZKPs are applied to high-risk situations, for example when used to protect medical data or to ensure transaction correctness in financial systems. We, therefore, consider the knowledge of the security models and assumptions to be vital knowledge when considering implementing NIZKPs, which we expect is the case for most people reading this work.

The third reason we provide for the importance of having a primitive knowledge of the inner workings of the three NIZKP protocols lies in implementation mistakes that others made previously, which anyone not acquainted can easily repeat in the future. To this end, we provide a rudimentary, non-exhaustive, overview of past vulnerabilities in NIZKP protocol implementations. For each vulnerability we describe the problem, state the affected protocols, and provide an example of an implementation where the vulnerability surfaced. Most importantly, we also provide a description on either how the vulnerability was resolved, or how the vulnerability can be prevented.

Because, as with all kinds of cryptographic algorithms, a sound implementation does not necessarily prevent security problems induced by a bad implementation, we express the essence for anyone implementing zero-knowledge proofs to have a minimum level of knowledge on the mathematical and cryptographic background of the protocol. This benefits the security of the application and consequently benefits the application users indirectly. Given that, from the above statements, we hypothesize that knowledge of the NIZKP protocol primitives is beneficial, especially for readers less well-versed on the topic, we include a concise overview of said primitives in this chapter.

4.2 Mathematical primitives: zk-SNARK

The goal of this section is to give an overview of the technologies in each NIZKP protocol, briefly explain the mathematical foundations those techniques are built on, and describe how these techniques are combined to form parts of the NIZKP protocol. In this specific section, we look at the primitives of the zk-SNARK protocol.

4.2.1 Polynomials

A polynomial P of degree d over $\mathbb{F}_p$ has the form $P(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_d \cdot x^d$ for coefficients $c_0, \ldots, c_d$. This means we can evaluate P at point s such that $P(s) = c_0 + c_1 \cdot s + c_2 \cdot s^2 + \cdots + c_d \cdot s^d$. For someone that knows P, $P(s)$ is a linear combination of $1, s, \ldots, s^d$. This linear combination is the weighted sum, and the coefficients $c_0, \ldots, c_d$ are the weights for $P(s)$.

4.2.2 Bilinear pairings

A bilinear pairing is a pairing between elements of two cyclic groups G_1, G_2 which map to a third cyclic group G_T when multiplied, i.e. $e : G_1 \times G_2 \rightarrow G_T$ [56]. More specifically:

- Let F_q be a finite field over prime q.
- Let G_1, G_2 be two additive cyclic groups of prime order q.
- Let G_T be a multiplicative cyclic group of prime order q.

Then a pairing is a mapping $e : G_1 \times G_2 \rightarrow G_T$ that satisfies:

- Bilinearity, i.e $P \in G_1, Q \in G_2 : e(aP, bQ) = e(P, Q)^{ab}$ for all $a, b \in F_q^*$.
- Non-degeneracy, i.e. $e \neq 1$.
- Computability, i.e. an efficient algorithm exists to compute e.

4.2.3 Homomorphic hidings

A homomorphic hiding is a weaker notion of a computationally hiding commitment scheme. It constitutes a commitment scheme without the random value used in commitment schemes. A homomorphic hiding $E(x)$ of number x satisfies [53]:

- Given $E(x)$ it is hard to find x (for most x).
- Different inputs give different outputs, i.e. if $x \neq y$, then $E(x) \neq E(y)$.
- Given $E(x)$ and $E(y)$, the homomorphic hiding of arithmetic expressions of x and y can be calculated. E.g. $E(x + y)$.

4.2.4 Discrete logarithm problem

A discrete logarithm is an integer x such that $g^x = y$ given an element $y \in G$, where g is a generator of cyclic group G of order n [61]. The Discrete Logarithm Problem (DLP) then constitutes the problem that, with some exceptions, there is currently no known general way to efficiently compute a discrete logarithm.

4.2.5 Elliptic curve cryptography

Elliptic Curve Cryptography (ECC) is a cryptography approach that uses elliptic curves over finite fields. The security of ECC depends on the ease of calculating the multiple x of a base point P on a curve to get a new point $R = xP$ and the inability to reverse this to find x given R and P. This means that the security of ECC depends on a form of the DLP, specifically the Elliptic Curve Discrete Logarithm Problem (ECDLP) [68].

Operations Elliptic curves allow to perform addition, doubling, and multiplication operations [80]. To explain these three operations we will use the basic elliptic curve $y^2 = x^3 + ax + b$. Note that all operations are performed modulo the size of the used field, e.g. mod 23 for $\mathbb{F}_{23}$.

- **Point addition**: To add two points $P = (x_p, y_p)$ and $Q = (x_q, y_q)$ together into $R = (x_r, y_r)$, we first determine the slope of the line between the two points $s = (y_q - y_p)/(x_q - x_p)$. We then get the coordinates of $R = (x_r, y_r)$ by setting $x_r = s^2 - x_p - x_q$ and $y_r = s(x_p - x_r) - y_p$.
- **Point doubling**: To add two points P with the same coordinates (called coincident) together, i.e. double the point to get $2P$, then there is no straight line through the two points. Instead, we can use the tangent of the point for the slope by taking the derivative of the curve. The slope then becomes $s = (3(x_p)^2 + a)/(2 * y_p) = (3(x_p)^2 + a) * (2 * y_p)^{-1}$. Note that $(2 * y_p)^{-1}$ here is the modular inverse of $2 * y_p$ under the modulo according to the used field. We can then get the coordinates of $R = (x_r, y_r)$ again by setting $x_r = s^2 - x_p - x_q$ and $y_r = s(x_p - x_r) - y_p$.
- **Point multiplication**: For the multiplication of a single point P, e.g. $3P$, we can use a combination of point addition and point doubling, called the double-and-add method. This means that $3P$ becomes $2P + P$, which we can calculate using the methods of addition and doubling.

4.2.6 Quadratic arithmetic program

To generate a zk-SNARK proof for any computation, we first need to transform this arithmetic circuit for this computation into a polynomial. This is possible using a Quadratic Arithmetic Program (QAP). To generate a QAP we first have to transform the computation into an arithmetic circuit and then in a Rank-One Constraint System (R1CS). Then we can transform the R1CS into a QAP to use in the zk-SNARK proof.

Arithmetic circuit An arithmetic circuit in our case is a directed acyclic graph consisting of gates that represent the arithmetic operations. We can create an arithmetic circuit by flattening the computation so that each gate has two inputs. We show an example arithmetic circuit for $x^3 + x + 5$:

$$o_1 = x \cdot x \tag{4.1a}$$
$$o_2 = g_1 \cdot x \tag{4.1b}$$
$$o_3 = g_2 + x \tag{4.1c}$$
$$o_4 = g_3 + 5 \tag{4.1d}$$

We then create a solution vector that includes all inputs and the output. For Equation 4.1 we get $S = [1, x, o_4, o_1, o_2, o_3]$, where 1 represents the constants. The solution vector S is also called a witness.

R1CS A R1CS represents an arithmetic circuit using four vectors. The vectors a, b, c represent the left inputs, right inputs, and outputs respectively, while vector S represents the solution polynomial. A valid solution polynomial S must satisfy

$$(a \cdot S) \cdot (b \cdot S) = (c \cdot S) \tag{4.2}$$

Note that in Equation 4.2 the multiplication of the vectors is a dot product. From Equation 4.2 we can see that to obtain the R1CS for the arithmetic circuit we should represent a, b, c as a mapping of S such that only the required values remain after the dot product with S. This gives us the following vectors at each gate in the arithmetic circuit from Equation 4.1:

$$a_1 = [0, 1, 0, 0, 0, 0] \tag{4.3a}$$
$$b_1 = [0, 1, 0, 0, 0, 0] \tag{4.3b}$$
$$c_1 = [0, 0, 0, 1, 0, 0] \tag{4.3c}$$

$$a_2 = [0, 0, 0, 1, 0, 0] \tag{4.4a}$$
$$b_2 = [0, 1, 0, 0, 0, 0] \tag{4.4b}$$
$$c_2 = [0, 0, 0, 0, 1, 0] \tag{4.4c}$$

For gate 1 (Equation 4.3) and gate 2 (Equation 4.4) we multiply by using the multiplication between the two dot products on the left-hand side of Equation 4.2.

$$a_3 = [0, 1, 0, 0, 1, 0] \tag{4.5a}$$
$$b_3 = [1, 0, 0, 0, 0, 0] \tag{4.5b}$$
$$c_3 = [0, 0, 0, 0, 1, 0] \tag{4.5c}$$

$$a_4 = [5, 0, 0, 0, 0, 1] \tag{4.6a}$$
$$b_4 = [1, 0, 0, 0, 0, 0] \tag{4.6b}$$
$$c_4 = [0, 0, 1, 0, 0, 0] \tag{4.6c}$$

For gate 3 (Equation 4.3) and gate 4 (Equation 4.4) we apply addition by using the dot product of one of the vectors with the solution vector (Equation 4.2).

Finally we can combine Equation 4.3, Equation 4.4, Equation 4.5, and Equation 4.6 to get three matrices L, R, O.

$$L = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.7a}$$

$$R = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.7b}$$

$$O = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{4.7c}$$

QAP We can now transform the R1CS to a QAP, which uses polynomials instead of dot products. To perform this transformation, we use either a Lagrange interpolation [76] or a considerably more efficient fast Fourier transform [116] approach to find a polynomial that goes through a specific set of points. We transpose the matrices from Equation 4.7 to find the coefficients for a polynomial that goes through the points (x, y), where x is the row number (starting at 1) of the transposed matrix and y is the value.

$$Lp = \begin{pmatrix} -5.0 & 9.166 & -5.0 & 0.833 \\ 8.0 & -11.333 & 5.0 & -0.666 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ -6.0 & 9.5 & -4.0 & 0.5 \\ 4.0 & -7.0 & 3.5 & -0.5 \\ -1.0 & 1.833 & -1.0 & 0.166 \end{pmatrix} \tag{4.8a}$$

$$Rp = \begin{pmatrix} 3.0 & -5.166 & 2.5 & -0.333 \\ -2.0 & 5.166 & -2.5 & 0.333 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix} \tag{4.8b}$$

$$Op = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ -1.0 & 1.833 & -1.0 & 0.166 \\ 4.0 & -4.333 & 1.5 & -0.166 \\ -6.0 & 9.5 & -4.0 & 0.5 \\ 4.0 & -7.0 & 3.5 & -0.5 \end{pmatrix} \tag{4.8c}$$

In our example, we obtain three sets of six degree-3 polynomials as shown in Equation 4.8. The first polynomial of Lp is $0.833x^3 - 5x^2 + 9.166x - 5$.

From Equation 4.2 we define target polynomial T

$$T = (Lp \cdot S) \cdot (Rp \cdot S) - (Op \cdot S) \tag{4.9}$$

The QAP now allows the verification of all constraints at once using a polynomial Z.

$$Z = (x-1)(x-2)(x-3)(x-4) \tag{4.10}$$

The solution is valid if T is divided by Z without a remainder, i.e. when there exists a polynomial H such that $T = H \cdot Z$.

Additionally, we provide a succinct version of the mathematical process for generating a zk-SNARK arithmetic circuit proof in Appendix A.

4.3 Mathematical primitives: zk-STARK

The goal of this section is to give an overview of the technologies in each NIZKP protocol, briefly explain the mathematical foundations those techniques are built on, and describe how these techniques are combined to form parts of the NIZKP protocol. In this specific section, we look at the primitives of the zk-STARK protocol.

4.3.1 Polynomials

See subsection 4.2.1.

4.3.2 Cryptographic hash functions

Cryptographic hash functions are algorithms that perform a relatively easy-to-compute one-way mapping of an arbitrary length input to a fixed length output called a hash value. Furthermore, a cryptographic hash function H needs to have the following properties [46] [71]:

- Pre-image resistant: for any hash value h it is computationally difficult to find input m such that $H(m) = h$. This makes it so that anyone who is only given the hash value h should not be able to reverse it to find the used input value m.
- Second pre-image resistant (weak collision resistant): for a given input m_1, it is computationally difficult to find another input m_2 such that $H(m_1) = H(m_2)$. This makes it unlikely that two different inputs give the same hash value output.
- Collision resistant (strong collision resistant): it is computationally difficult to find two different inputs m_1, m_2 such that $H(m_1) = H(m_2)$. This makes it hard for anyone to find two different inputs that map to the same hash value h. The difference is that collision resistance defines it computationally hard to find m_2 such that $H(m_1) = H(m_2)$ for any m_1, whereas second pre-image resistance specifies this for specific m_1.

Because of their properties, cryptographic hash functions can be used to verify data integrity [108], generate pseudo-randomness from an input [87], or function as a binding commitment [44].

4.3.3 Merkle trees

A Merkle tree is a commitment scheme that uses cryptographic hashes of committed values in a tree shape. In a Merkle tree, the value of each node is the hash value output of the combined (i.e. concatenated) child nodes as input [79].

To create a Merkle tree, we split the data into i blocks d_i where ideally $i = 2^n$ for some n. The i leaf nodes l_i are then each assigned hash value $H(d_i)$. The $j = i/2$ nodes n_i above the leaf nodes are subsequently assigned the hash value of $H(l_{j*2}||l_{j*2+1})$. This process repeats until the root node is reached and the Merkle tree is complete. The Merkle tree can then be represented by the root node hash value.

To verify a data block in the Merkle tree, the prover reveals the single data block value and additionally provides the hash values required for the authentication path, i.e. the hash value of each other child node connected to all nodes on the way up to the root. This means that the verifier can hash the data block themselves, then recursively calculate the hash of the parent node from the child node hash they calculated concatenated with the received hash of the other child node connected to the parent node. The verifier repeats this step until eventually the root is reached, at which point the calculated hash value should correspond to the actual root hash value.

We now provide an example of a Merkle tree for the given sentence "This is an example." split into four data blocks: ["This", "is", "an", "example."]. Using the SHA-1 hash algorithm, the Merkle tree then becomes:

- **Root node**: `96C3C30ACF7AB04C3E06A88DB4C881711419736E`
- **Layer 1 nodes**: [`3AAAFF4FA6ADDC7363C51553D49E9473D41E9C8B`, `B22A0716345D6354F102AF8AD58573FA83E9417D`]
- **Leaf nodes**: [`7971E6A051104074FDAE0F02322417B6EB5695A2`, `B47F363E2B430C0647F14DEEA3ECED9B0EF300CE`, `DE73EAC0C305038F0437BC6A1F994A5A4379ED28`, `9133AB60CAA1C7BE379F21F1BA2968365A54BF36`]

A verifier can then verify a claim from a prover, e.g. that the second data block in the Merkle tree is the word "is". To verify this, the prover provides the data block "is" alongside the hash values required for the authentication path, i.e. the first leaf node `7971E6A051104074FDAE0F02322417B6EB5695A2` and the layer 1 node `B22A0716345D6354F102AF8AD58573FA83E9417D`. The verifier then SHA-1 hashes the word "is" to `B47F363E2B430C0647F14DEEA3ECED9B0EF300CE`, then calculates the parent layer 1 node hash value `3AAAFF4FA6ADDC7363C51553D49E9473D41E9C8B` by concatenating the calculated and provided leaf node hashes and hashing it. The verifier finally calculates the root node hash value `96C3C30ACF7AB04C3E06A88DB4C881711419736E` by hashing the concatenation of the calculated and provided layer 1 node hashes as input. If this calculated root node corresponds to the Merkle tree root node, then the verifier accepts the claim from the prover and knows that the second data block indeed contains the word "is". The verifier, however, still does not know the other data blocks in the Merkle tree.

4.3.4 Fast Fourier transform

The Fast Fourier Transform (FFT) is a fast and recursive way to apply the Discrete Fourier Transform (DFT) [40]. For any degree D polynomial, the coefficient representation uniquely defines a polynomial through $D+1$ coefficients, whereas the value representation does so through $D+1$ points on a polynomial. The reason for this is that a degree D polynomial can be uniquely defined by $D+1$ points on the curve [73]. Whereas a regular

DFT uses a matrix multiplication to convert from one representation to another. Such a matrix, however, contains several symmetries, which FFTs take advantage of to reduce the total number of calculations. This gives the FFT a complexity of $O(n \log_2(n)$, instead of $O(n^2)$ for the regular DFT. As a result, we can use the FFT to efficiently convert between the two representations of a polynomial. The transformation from the value representation to the coefficient representation is specifically called interpolation.

For the prerequisites, we first use Euler's formula $e^{ix} = \cos(x) + i\sin(x)$ [115] to get the n n^{th} roots of unity $e^{(2\pi ik)/n}$ for $k \in [0, 1, \ldots, n-1]$ [118] and the principal n^{th} root of unity $\omega_n = e^{(2\pi i)/n}$ [117].

The FFT pseudocode algorithm for Polynomial $P(x)$ with coefficients $[c_0, c_1, \ldots, c_{n-1}]$ and principal roots $[\omega^0, \omega^1, \ldots, \omega^{n-1}]$ is then:

- If $n = 1$, return $P(1)$.
- Otherwise, define an even side as $P_e(x^2)$ using $[p_0, p_2, \ldots, p_{n-2}]$ and $[\omega^0, \omega^2, \ldots, \omega^{n-2}]$, and an odd side as $P_o(x^2)$ using $[p_1, p_3, \ldots, p_{n-1}]$ and $[\omega^0, \omega^2, \ldots, \omega^{n-2}]$. This gives $y_e = [P_e(\omega^0), P_e(\omega^2), \ldots, P_e(\omega^{n-2})]$ and $y_o = [P_o(\omega^0), P_o(\omega^2), \ldots, P_o(\omega^{n-2})$. This is the recursive step that keeps splitting into two parts until $n = 1$.
- Assemble and return the result $y = [P(\omega^0), P(\omega^1), \ldots, P(\omega^{n-1})]$ using $P(\omega^j) = y_e[j] + \omega^j y_o[j]$ and $P(\omega^{j+n/2}) = y_e[j] - \omega^j y_o[j]$ for $j \in [0, 1, \ldots, n/2 - 1]$.

The pseudocode FFT algorithm described above uses $\omega = e^{(2\pi i)/2} = e^{\pi i}$ to transform a polynomial from coefficient representation to value representation. The algorithm can also be used to transform a polynomial from value representation to coefficient representation, i.e. perform interpolation, by using $\omega = e^{-\pi i}/n$.

4.3.5 Reed-Solomon codes

The Reed-Solomon (RS) code is an error correcting code algorithm which uses finite field arithmetic, specifically Galois fields, and polynomials to encode redundancy on a message. In Reed-Solomon codes, the parameter k is the number of symbols in the message, while n is the number of digits in the final codeword. This means that the codeword contains $t = n - k$ redundant symbols, which allows RS codes to detect up to the same number of incorrect symbols. RS codes can simultaneously correct up to $(n-k)/2$ (rounded down to the nearest integer) symbols to obtain the original k-symbol message. The minimum Hamming distance [67] between codewords for different messages is $n - k + 1$.

The basic idea of the RS codes algorithm is to encode a message in a polynomial. The k message symbols are used to define a polynomial of degree $k-1$, with each message symbol being a coefficient of the polynomial. Specifically, message $m = \{m_0, m_1, \ldots, m_{k-1}\}$ give polynomial $p_m(a) = m_0a^0 + m_1a^1 + \cdots + m_{k-1}a^{k-1}$. The encoded codeword c is then the polynomial evaluated at $n \leq q$ distinct points $\{a_0, a_1, \ldots, a_{n-1}\}$ in finite field $\mathbf{F}$ with q elements, i.e. $c = \{p_m(a_0), p_m(a_1), \ldots, p_m(a_{n-1})\}$, where a common choice for a is the index such that $a_i = i$. There also exists a version named the "systematic encoding procedure", where the the polynomial p_m is set such that $p_m(a_i) = m_i$ for all $i \in [0, \ldots, k-1]$. The general idea however is the same.

The error detection and correction are performed by interpolating a polynomial from the codeword using e.g. the Berlekamp-Massey algorithm [20]. Then the errors can be located by using for example the Chien search algorithm [36] on the polynomial, and corrected by using e.g. Forney's algorithm [52].

4.3.6 Fast Reed-Solomon interactive oracle proofs of proximity

The Fast Reed-Solomon Interactive Oracle Proofs of Proximity (FRI) is an Interactive Oracle Proof of Proximity (IOPP) protocol that uses RS codes as a means to prove proximity. As described in subsection 4.3.4, a polynomial of degree D can be uniquely defined by $D+1$ coefficients or points on the polynomial. This means that we can prove knowledge of a degree $< D$ polynomial by letting the verifier choose D random points. The prover then provides the coordinates for these points such that the verifier can interpolate to recover the unique degree $< D$ polynomial passing through them. The verifier subsequently asks the prover for some more randomly sampled points and then verifies that those are also on the interpolated polynomial. However, this solution is not succinct. We want to verify a degree $< D$ polynomial by asking for fewer than D points. FRI is an algorithm that can do this.

The goal of FRI is for the prover to convince the verifier that a function f is close, i.e. within a small defined Hamming distance [67] margin, to a low-degree polynomial $p(X)$ of a specified maximum degree [66]. This is done in two main phases [12]:

- **Commit phase**: This phase consists of r rounds of splitting the polynomial from the previous round into several parts according to the reduction factor. This is called the split-and-fold technique, which similarly to FFT solves the problem faster by recursively reducing the problem size. For a reduction factor of 2, the polynomial function is split into two parts: even and odd. The verifier then samples a random challenge which it sends to the prover. This challenge the prover uses to calculate a polynomial with a lower degree by interpolating the function. The prover thereafter calculates the next function using the received challenge and the polynomial and sends it to the verifier. This process continues until the last round, where instead the interpolated function polynomial, which should now have a low degree, is revealed by the prover.

- **Query phase**: In this phase, the verifier asks for random points from the domain in their specification. These points are subsequently used to check the consistency of each reduction step in the commit phase. The commit phase consists of $S \geq 1$ rounds, in which the verifier samples a random point x_0 from the defining domains of the oracles each, and then calculates $x_1, \ldots, x_r$ recursively. Finally, the verifier ensures that the values are consistent with the provided reduction steps.

For a more detailed mathematical background, we refer to the original FRI paper by Ben-Sasson et al. [12]. For an explanation more succinct than the original paper yet more involved than ours, we refer to a summary on FRI by Haböck [66].

On a final note, it is important to know that FRI by itself does not include zero-knowledge, which means that it has to be implemented on an application level [66]. This will be a significant fact for chapter 5 relating to the utilized zk-STARK library.

Polynomial commitment scheme FRI can be transformed into a polynomial commitment scheme [66]. The FRI polynomial commitment scheme consists of three phases [23]: generate, commit, and open. For the polynomial commitment schemes using FRI, these steps involve the following operations:

- **Generate**: The polynomial is converted to a RS codeword where the message length k is polynomial degree $d + 1$.

- **Commit**: The prover commits to the evaluation of the polynomial over the entire domain. In zk-STARK, this uses the Merkle trees described in subsection 4.3.3 [14].
- **Open**: The prover and verifier perform a batched FRI argument. This involves batching the functions into linear combinations and performing FRI on them.

To keep this section succinct, we refer to other works for a more detailed mathematical background on polynomial commitment schemes. A work by Boneh et al. [23] describes polynomial commitment schemes in general, whereas a summary on the FRI algorithm by Haböck [66] contains a section on the polynomial commitment scheme used in FRI specifically.

In summary, the RS codeword represents the polynomial which is claimed to be of a low degree. In more detail, for a codeword of length N and polynomial of maximum degree d, the polynomial is $f(X) = \sum_{i=0}^{d} c_i X^i$. The polynomial is then evaluated at N distinct points. Whereas the prover knows the codeword, the verifier only knows the Merkle commitment root and the chosen leafs, which it can use to validate the provided proof [3].

4.4 Mathematical primitives: Bulletproofs

The goal of this section is to give an overview of the technologies in each NIZKP protocol, briefly explain the mathematical foundations those techniques are built on, and describe how these techniques are combined to form parts of the NIZKP protocol. In this specific section, we look at the primitives of the Bulletproof protocol.

4.4.1 Discrete logarithm problem

See subsection 4.2.4.

4.4.2 Elliptic curve cryptography

See subsection 4.2.5.

4.4.3 Pedersen commitments

A Pedersen commitment [99] is a commitment scheme with special mathematical properties:

- Proving additive equalities (modulo the used group order).
- Hiding property even for computationally unbounded adversaries.

’Regular’ form Pedersen commitments use a public multiplicative group $(G, \cdot)$ of large order q in which the discrete logarithm is hard. G_q is the unique subgroup of $\mathbb{Z}_p^*$ or order q. Furthermore, there are two random public generators g and h of G_q. To create a Pedersen commitment, choose a secret random $r \in \mathbb{Z}_q$ such that the message $m \in \mathbb{Z}_q$, then calculate the commitment with $\mathcal{C}(m, r) = g^m \cdot h^r$. To open the commitment, reveal message m and randomness r. Anyone can then verify whether the commitment committed to m using the commitment calculation $\mathcal{C}(m, r) = g^m \cdot h^r$. If randomness $r = 0$, then the commitment is binding, meaning that the committer cannot open a commitment to $m' \neq m$ because of the DLP, but not hiding, meaning that the commitment may reveal information about m.

For other values of r, i.e. when randomness is used in the commitment, the commitment is both binding and hiding. Here, the hiding property means that the commitment does not reveal any information about m.

Elliptic curve form The elliptic curve form of a Pedersen commitment differs from the 'regular' form by using different parameters:

- G is the publicly agreed generator of the elliptic curve.
- H is another curve point vector, calculated with $H = qG$ such that nobody knows the discrete logarithm q.
- C is the commitment as a point on the elliptic curve, calculated using $C = aG + rH$.
- a is the committed value.
- r is the used randomness.

Note that, since we deal with points on an elliptic curve, the addition and multiplication operations must be performed as ECC point addition and multiplication respectively. We previously described how that works in subsection 4.4.2. The elliptic curve form of a Pedersen commitment is homomorphic as long as we set the randomness of the combined commitment to be equal to the sum of the randomness in the individual commitments. The reason for this is the calculation $C(r_1, a_1) + C(r_2, a_2) = r_1H + a_1G + r_2H + a_2G = (r_1 + r_2)H + (a_1 + a_2)G = C(r_1 + r_2, a_1 + a_2)$.

Vector Pedersen commitment The vector Pedersen commitment is a version of the Elliptic curve form Pedersen commitment that works with vectors. To create such commitment, we randomly generate a vector of generators $\mathbf{g}$ and generator h from the group $\mathbb{G}$ of order p: $\mathbf{g} = (g_1, g_2, \ldots, g_n), h \leftarrow \mathbb{G}$. Then to commit to a vector of values $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, compute the commitment as $C = h^r g^x = h^r \prod_i g_i^{x_i} \in \mathbb{G}$ where r is the used randomness.

4.4.4 Inner product argument

The basis of the Bulletproof protocol consists of the inner product argument. Specifically, the protocol constructs an inner product polynomial of two vector polynomials such that evaluating the inner product polynomial at a given x provides the same result as separately evaluating the vector polynomials at x, and afterward taking the inner product [33]. However, this by itself does not constitute a range proof that the Bulletproof protocol is known for. In their work [33], Bünz et al. first reduce the complexity of the inner product proof idea as introduced by Bootle et al. [24]. Subsequently, they provide constructions to create efficient range proofs using their improved inner product argument and specific traits of Pedersen commitments. Finally, the Bulletproof paper [33] also describes a way to use the Bulletproof protocol for arithmetic circuits, for which they again use their improved inner product. Their method features improved efficiency in creating arbitrary arithmetic circuit proofs compared to the method described by Bootle et al. [24]. To summarize, independent of whether the Bulletproof protocol is used to generate range proofs or arithmetic circuit proofs, it internally uses an efficient inner product argument to create that proof.

Additionally, we provide the mathematical process of generating a Bulletproof arithmetic circuit proof in Appendix B.

4.5 Security models and assumptions

In this section, we discuss the security models and assumptions of the NIZKP protocols. It aims to explain the basic cryptographic assumptions and prerequisites in each protocol and the security assumptions under which each protocol is deemed secure. This also includes a brief identification of the threat models and adversaries that each protocol does or does not intend to protect against.

4.5.1 zk-SNARK

Security models

- **Zero-knowledge property** A proof system is zero-knowledge if, by providing a proof, the prover does not reveal any information other than the information inherent in the truth of the statement that it proves [59]. In other terms, zero-knowledge means that the generated proof does not leak information about the inputs used to generate the proof [19]. This kind of zero-knowledge is defined as statistical or perfect zero-knowledge [64], since zero-knowledge would not always hold in reality if unbounded resources were available (unbounded by polynomial-time). When zero-knowledge holds only under the assumption of polynomially bounded resources, then this is instead defined as computational zero-knowledge [64]. Since one of the defining characteristics of NIZKPs is that the prover can prove knowledge without revealing this knowledge to the verifier, NIZKPs need to implement some form of zero-knowledge.

- **Succinctness** A proof system is succinct if it allows for efficient verification of a proof. More specifically, when a verifier can verify a nondeterministic polynomial-time computation in a fraction of the time it took the prover to run the computation itself [16]. The goal of succinctness is to allow a verifier to verify the correctness of a computation using few resources [19], which can be useful to delegate computations to parties with more compute resources without the requirement for ultimate trust in those parties. By default, proof systems that achieve succinctness can only be assumed to be computationally sound, meaning that they are only secure against bounded-size adversaries [16].

- **Non-interactivity** A proof system is non-interactive if it does not require interaction between the prover and the verifier. This means that the prover can generate the proof on their own, even when offline, and later distribute the proof with the required parameters to a verifier for verification [64]. Traditional interactive proof systems can be transformed into a NIZKP using the Fiat-Shamir heuristic [51]. To remove the interaction aspect, the Fiat-Shamir transform replaces the random challenges that the verifier usually generates in an interactive proof system with the result of the hash function on the input and outputs up to that point [74]. The prover cannot easily cheat this process, since the hash function outputs pseudorandomness based on the input [87]. An additional benefit provided by non-interactive proof systems is that many verifiers can independently verify the same proof since there is no interaction between a prover and a single verifier. The proof systems implementing non-interactivity are dependent on the security of the Fiat-Shamir transform, which researchers have shown to be secure under the random-oracle model [101]. Still, the (in)security of the heuristic has been the subject of several works [19] [42] [70] that discuss situations in which the heuristic cannot be assumed to be secure. In zk-SNARKs, the non-interactive property is furthermore enabled by the use of a Common Reference String (CRS), first introduced by Blum et al. under the name of a common random string [22]. Such CRS is generated during a trusted setup procedure (see "Trusted setup" in section 4.5.1).

Cryptographic assumptions and prerequisites

- **Pairing based cryptography** Pairing-based cryptography is one of the prerequisites of zk-SNARKs, as explained in subsection 4.2.2. In short, the idea of pairing-based cryptography is to map elements in two separate groups to an element in a common third group [56]. This process helps with the security and computability of some transformations in zk-SNARKs. A bilinear mapping consists of two parts: an elliptic curve and a pairing function [89]. For bilinear pairings, this means that the security depends on the same security assumption as ECC (see subsection 4.2.4. The security assumption of pairing-based cryptography is specifically known as the weaker and less studied Bilinear Diffie-Hellman Problem (BDHP), which breaks when either of the DLP or BDHP can be solved [89].

- **Trusted setup** The trusted setup in zk-SNARK generates a Common Reference String (CRS). The CRS is the set of parameters that are used to create proofs and perform verifications in the zk-SNARK process [100]. The CRS is, in turn, generated using secret random variables which are also called toxic waste. The reason for this dubious name comes from the requirement to properly destroy these variables after use, as these variables could otherwise be abused to generate undetectable false proofs [100]. The simplest way to perform a trusted setup is by asking a single trusted third party to perform the procedure. However, this is also a dangerous option since it assumes the trusted third party is not malicious and properly disposes of the toxic waste. Because this insecurity limits the applicability of NIZKPs that require trusted setup, researchers designed other trusted setup schemes that do not depend on a single trusted party. Such schemes include multi-party protocols to distribute the setup procedure over multiple parties, where no single party knows the complete witness [15]. This means that the security of the trusted setup now assumes the security of the Multi-Party Computation (MPC) protocol and that at least a single honest party participates in the MPC. Specifically, this means the MPC trusted setup is not secure when all participants in the protocol collude [15].

Threat model and adversary capabilities

- **Soundness** Soundness in a proof system is the property that a prover cannot convince a verifier to accept a proof of a false statement, at least not with a high probability [17]. This means that if a verifier accepts a proof, then the prover should, except for a negligible probability, have the knowledge to create a proof for the corresponding statement. Soundness is therefore part of the threat model, to ensure that the protocol mathematically guarantees a negligible probability that the prover creates a valid proof while the statement is not true. In addition, the prover should not be able to craft their inputs in a way that significantly improves the probability of generating a false proof, thereby breaking the soundness assumption. In practice, the soundness of the protocol is highly dependent on the proper execution of the trusted setup to ensure that the secret randomness is destroyed and not leaked [15]. See "Trusted setup" under "Cryptographic assumptions and prerequisites" in section 4.5.1.

- **Zero-knowledge property** The zero-knowledge property in proof systems is the property that, for any true statement, a prover can convince a verifier of a statement without giving away any information used in creating the proof except for the information of the statement itself [17]. See also "Zero-knowledge property" under "Security models" in section 4.5.1. As discussed in that section, perfect zero-knowledge is not a realistic assumption in practice. This means that the threat model should include the zero-knowledge property to ensure that the alternative assumption of computational zero-knowledge holds. If an adversary could break the computational zero-knowledge, then this would mean the malicious verifier could obtain auxiliary information used while generating the proof. This constitutes a security and privacy problem since the prover assumed this information would not become known, meaning that users cannot trust the NIZKP protocol. Examples of situations where this is particularly troublesome are when the data it leaks are privacy or business-sensitive information, or the proof inputs include secret knowledge such as private keys. It is these data types are, however, that are often involved in the creation of NIZKPs.

- **Setup assumptions** As discussed under "Trusted setup" in section 4.5.1, the proper execution of the trusted setup depends on either a single trusted party that should destroy the toxic waste, or on the security of a MPC protocol where at least a single honest party must have joined the trusted setup MPC. The threat model here consists of the adversary being able to control the trusted setup procedure to obtain the toxic waste, which would allow the adversary to create convincing false proofs [15]. The use of a MPC for the trusted setup can reduce this threat, yet even then one should consider whether an adversary could somehow prohibit honest parties from joining the MPC or convince all participants to collude. Another separate part of the threat model considers the possibility of a bad trusted setup protocol, in which the setup leaks the secret parameters. This threat model is especially applicable to MPC trusted setups since these are provably more complex. Such a trusted protocol leak has happened in practice where the transcript of a MPC trusted setup leaked the secret parameters [113]. We further detail this vulnerability in subsection 4.6.2. The final consideration in the threat model is that the CRS, generated by the trusted setup, should be disclosure-free such that nobody can extract useful information from it [63]. If the CRS is not disclosure-free, then an adversary can use the extracted information to create false yet valid proofs.

4.5.2 zk-STARK

Security models

- **Zero-knowledge property** See "Zero-knowledge property" under "Security models" in section 4.5.1.

- **Scalability** A proof system is (fully) scalable if the run durations for both the prover and the verifier are quasi-linear in the input [14]. Scalability intends to retain the performance in real-world systems implementing NIZKPs when they grow larger, by ensuring that the verifier complexity is quasi-linear [14]. Scalability is an important aspect in the security of zk-STARK because it allows systems to scale while utilizing zk-STARKs in their application to provide the security and privacy aspects. This is different from protocols that depend on the DLP, including the zk-SNARK [64] and Bulletproof [24] protocols, where the verifier complexity in the DLP approach is not quasi-linear in time even though the communication complexity is logarithmic. This can lead to concessions in the use of NIZKPs, which in turn can reduce the security of systems to retain scalability. By not depending on non-scalable problems such as the DLP, the zk-STARK protocol can achieve full scalability while providing all other benefits of a NIZKP.

Cryptographic assumptions and prerequisites

- **Polynomial commitment scheme** Commitments allow a prover to commit to a certain value by binding it to use this value while generating the proof. The commitment hides the value when the prover provides it to the verifier [72]. The verifier can subsequently validate that the value, on which the prover generated the proof, is consistent with the value in the commitment. A polynomial commitment scheme provides a commitment for a polynomial of a certain degree, which limits the prover to perform the proving steps on the actual polynomial of a bounded degree [23]. This is an important aspect of the zk-STARK protocol, since otherwise, an adversary could easily commit to some random values that are not on any low-degree polynomial that would pass the internal checks anyways [34]. Doing so would break the security of the protocol. For this reason, the zk-STARK protocol assumes and requires the availability of a polynomial commitment scheme that is both hiding, it does not reveal the committed value and binding, the prover cannot open the commitment to a different value than was committed to [44]. See section 4.3.6 for more details.

- **Post-quantum security** The authors of the original zk-STARK paper [14] mention in their work that their protocol is post-quantum secure. Post-quantum security means that the protocol should still be secure even when large-scale quantum computers become feasible. To obtain this post-quantum security, the authors relied on the security assumptions of the underlying techniques of collision-resistant hash functions and random functions [14]. Collision-resistant hash functions do not depend on any assumptions that are known to be quantum insecure such as the DLP, which Shor's algorithm breaks [109]. In general, while some algorithms exist that reduce the time complexity of collision-resistant hash functions using quantum computers, including Grover's algorithm [65], they only reduce the time complexity of an n-bit hash function to $\mathcal{O}(2^{n/2})$ or possibly $\mathcal{O}(2^{n/3})$ using Quantum Random Access Memory (qRAM) [45].

Threat model and adversary capabilities

- **Soundness** See "Soundness" under "Threat model and adversary capabilities" in section 4.5.1.

- **Zero-knowledge property** See "Zero-knowledge property" under "Security models" in section 4.5.1.

- **Quantum threats** Quantum threats are part of the zk-STARK threat model since it is vital to prevent quantum threats to obtain post-quantum security as discussed in "Post-quantum security" under "Cryptographic assumptions and prerequisites" in section 4.5.2. While currently no sufficiently powerful quantum computers exist to run Shor's algorithm in a way that makes factorization attacks feasible, quantum computers were demonstrated to factor up to only $N = 21$ as of 2021 [111], such computers could conceivably perform such attacks in the future. Especially when considering recent improvements on Shor's algorithm [105], it may be possible to break cryptographic algorithms that depend on the DLP somewhere in the future. By not using techniques that depend on the DLP, or other problems that are known to be vulnerable to quantum attacks, the zk-STARK protocol can remain secure in the presence of sufficiently powerful quantum computers. This assumption comes with the caveat that researchers may find new quantum algorithms in the future, which could sharply reduce the security of the techniques underlying zk-STARKs. Based on current cryptographic knowledge, though, this is an unlikely prospect [45]. If sufficiently powerful quantum computers becomes available to attack non-quantum secure protocols, then the initial cost of these machines may be prohibitively expensive. We expect, however, that cloud providers will provide access to such computers for fees that are a fraction of the purchasing price. For example, Amazon Web Services (AWS) already provides services for currently existing quantum computers [103]. This means that the future availability of sufficiently powerful quantum computers is not only a dangerous capability for e.g. state-sponsored actors, but also for adversaries for whom the benefits of performing quantum attacks outweigh the costs of using quantum computing cloud services. We argue this to be especially dangerous for high-value targets including financial systems. Unlike other protocols that are known to be quantum-insecure, however, the zk-STARK protocol aims to defend against the potential future threat of quantum computers by being post-quantum secure. This includes considering quantum threats in the threat model [14].

4.5.3 Bulletproofs

Security models

- **Zero-knowledge property** See "Zero-knowledge property" under "Security models" in section 4.5.1.

- **Conciseness** In the context of the Bulletproof protocol, conciseness is the idea that the proofs are much smaller than the size of the witness. I.e. for arithmetic circuits, the proof size is logarithmic in the size of the circuit [33]. This means that when the size of the input increases exponentially, the size of the proof only increases linearly. The importance of this aspect for the security model of Bulletproofs is that even proofs for large inputs remain small, which benefits systems that need to distribute the proofs over e.g. a Blockchain. Like succinctness in zk-STARK, this allows the usage of Bulletproofs in situations restricted by storage availability. Unlike in the zk-STARK protocol, however, the proof generation and verification times are linear in the input [33]. This means that it may be difficult to utilize Bulletproofs in systems that have to scale in the proof generation and verification time aspect. An additional benefit of the conciseness of the Bulletproof protocol is that the protocol allows for the aggregation of proofs [33]. This enables the logarithmic scaling of proof sizes by combining multiple proofs into a single proof. As a result of the logarithmic nature of the size of aggregated proofs, the combined proof has a lower space requirement than the total of the separate proofs [57]. The security model for this aggregation includes the usage of a MPC protocol to remove the requirement for involved parties to disclose the inputs they used to each other [33]. Besides the intricacies involved in the MPC, Bünz et al. proved that the security of aggregated proofs is similar to the non-aggregated variant. Specifically, "The aggregate range proof [...] has perfect completeness, perfect honest verifier zero-knowledge and computational witness extended emulation" [33].

Cryptographic assumptions and prerequisites

- **ECC** The Bulletproof protocol depends on the security of ECC, mostly for scalar multiplication of a point on the curve to a new point on the curve. Reversing this calculation in ECC is a known hard problem called the Elliptic Curve DLP (ECDLP) [68]. The security additionally depends on the assumption that the ECDLP is hard. While the DLP is well-studied and considered secure in the classical realm, it has been solved in the quantum realm using Shor's algorithm [109]. Therefore, the protocol assumes that quantum computers sufficiently powerful to run Shor's algorithm for large prime factors do not exist, which, for now, holds [111]. ECC can be configured with different curves [18] and security parameters [35]. In general, though, cryptographers describe the security level of cryptographic algorithms using the number of bits. For example, the elliptic curve secp256k1, used by the implementation described in the original paper, has a 128-bit security level [33].

Threat model and adversary capabilities

- **Soundness** See "Soundness" under "Threat model and adversary capabilities" in section 4.5.1.

- **Range proof** For the range-proof ability of the Bulletproof protocol, the threat model should include the potential capability for an adversary to cheat the proof and prove that a value is within the set limit when this is not the case. As described by Bünz et al. in the original Bulletproof work [33], range proofs are beneficial for many purposes including cryptocurrencies. For cryptocurrencies, however, the importance of security is clear in that it prevents adversaries from breaking the range proof used in transactions for monetary gain. Similarly, range proofs prevent the insertion of fake accounts with negative balances in proofs of solvency [33] [43]. Additionally, range proofs can be important to prove that a value is within a certain range where it could cause an over- or underflow vulnerability if it is outside this range. See subsection 4.6.1.

- **Elliptic curve security** Elliptic curves are an important part of Bulletproofs, as discussed in subsection 4.4.2 and "ECC" under "Cryptographic assumptions and prerequisites" in section 4.5.3. ECC is therefore important to include in the Bulletproofs threat model, which should at least ensure that the chosen ECC curves and parameters are secure and have a high enough n-bit security level. When opting to use low bit-level security instead, an adversary can use sufficiently powerful hardware to solve the DLP assumption within reasonable time [1] using specialized algorithms. Examples of such efficient algorithms include Pollard's rho and kangaroo algorithms [102]. Additionally, the threat model should include the future possibility of an adversary having the capability to use a sufficiently powerful quantum computer to use Shor's algorithm [109] to find an answer to an instance of the ECDLP problem. This would consequently break the security of a corresponding elliptic curve [77]. The final part of the threat model concerning elliptic curve security is the discussion of the potential capability for an adversary to include a backdoor. Specifically, the Dual EC DRBG algorithm was previously suspected to include such a backdoor, which eventually led to the removal of the algorithm from NIST's list of recommended algorithms [92]. While a backdoor is always a potential threat, the existence of such a backdoor is highly unlikely for well-researched algorithms that cryptographers consider secure [18]. Regarding this, we should note that the SafeCurves project does not consider the secp256k1 curve, used by the implementation described in the original Bulletproof paper [33], entirely safe [18]. They argue that the curve is not necessarily broken, but that the small number of discriminant values allows a speedup which makes proper security more complicated [106].

4.6 Vulnerabilities and limitations

In this section, we provide a non-exhaustive selection of common vulnerability types affecting NIZKP protocols. We list the affected protocols, provide a brief description of the vulnerability, some real-world examples, and a solution or prevention method to the vulnerability.

4.6.1 Arithmetic overflows and underflows

Affected protocols

- Bulletproofs
- zk-SNARK
- zk-STARK

Description Arithmetic overflows and underflows are vulnerabilities where numbers "wrap around" when they reach their maximum or minimum. For example, if we have a 32-bit positive number for values 0 to 4294967295, then when we subtract a number from another number such that the value would go below 0, the value wraps around and the remainder is subtracted from 4294967295. This example represents an arithmetic underflow. An arithmetic overflow is the opposite of this, where if we add some number to another number such that the value would go above 4294967295, then the value wraps around to 0 and the result becomes the remainder.

Example A simple example situation where this vulnerability can create problems is in financial transactions when someone spends more than their current balance. If the implementation design does not carefully consider this vulnerability by setting bounds, then users can abuse this to obtain an exceptionally large account balance.

Solution/prevention A solution to this vulnerability is therefore to properly ensure the bounds of numbers. For our simple example transaction, this would mean ensuring both the balance and the transaction amount are positive, and that the current balance is at least as much as the transaction amount to be subtracted. In general, one should prevent this vulnerability by constraining the input parameters in a way that ensures that arithmetic over or underflows are impossible.

4.6.2 Trusted setup leak

Affected protocols

- zk-SNARK

Description ZKP protocols that require a trusted setup, including zk-SNARK protocols, use secret variables in this setup. These variables, called the "toxic waste" of the trusted setup, need to be properly destroyed. Any adversary that gains access to these variables could generate false proofs that any verifier would accept. In practice, this trusted setup is often performed using Multi-Party Computation (MPC) to prevent anyone from knowing all "toxic waste" parameters.

Example The zk-SNARK protocol implementation used in ZCash contained a vulnerability in the trusted setup that exposed information on the "toxic waste" in the MPC transcript [113]. ZCash published this transcript after the trusted setup, meaning that any actor with access to the transcript could have potentially forged proofs.

Solution/prevention The vulnerability in ZCash was resolved by switching to a new proving system with a newly performed trusted setup that did not leak the same variables. The general way to prevent this vulnerability is to ensure the careful design of the trusted setup procedure. This should guarantee that the setup parameters are disposed of properly, such that they cannot leak.

4.6.3 Frozen heart vulnerability

Affected protocols

- Bulletproofs
- zk-SNARK
- zk-STARK

Description When the hash calculation in the Fiat-Shamir transform which makes a protocol non-interactive is not carefully applied to all inputs, then a malicious prover could more easily create fake proofs that a verifier accepts as valid [84]. Since all three NIZKP protocols use the Fiat-Shamir transform to make the protocol non-interactive, all three protocols are potentially vulnerable when using a wrong implementation.

Examples A version of this vulnerability was found in the original Bulletproofs paper [85]. To create a non-interactive variant of the Bulletproofs, the prover calculates the challenge received from the verifier in the interactive protocol. To do this, the implementation should compute a hash on all previous public values and commitments such that the prover cannot control the challenge value and abuse it to create a fake proof. The protocol description in the original Bulletproofs paper did not include the Pedersen commitments in the Fiat-Shamir transform. As a result, the used challenge values would be independent of the Pedersen commitment, meaning that any prover could easily compute challenge values that generate a false proof that a verifier would accept.

Several implementations of the Plonk zk-SNARK protocol were also affected by this vulnerability [86]. A probable cause for this was lacking clarity in the outline of the usage of the Fiat-Shamir transform in the Plonk paper [54].

Solution/prevention An implementation can prevent this vulnerability by ensuring that the hash calculation for the Fiat-Shamir transform includes all public inputs. This in turn ensures the secure execution of the Fiat-Shamir transform, which removes the ability to efficiently create false proofs.

4.6.4 Under-constrained circuits

Affected protocols

- Bulletproofs
- zk-SNARK
- zk-STARK

Description Under-constrained circuits are circuits that do not include sufficient input value checks for the circuit. A circuit is for example under-constrained when, according to expectations in the circuit design, an input value is outside the specified range. This vulnerability alternatively arises when the implementation expects that a value cannot be the identity or zero element, yet the circuit does not verify that the input conforms to this expectation.

Example A real-world example where this vulnerability surfaced was in the Aztec network [5]. The network used zero-knowledge proofs to prove that users spend a so-called "note" only once. The vulnerability in particular existed because the proof circuit expected the note index value to be 32-bit, while the input was not bound to this expectation. This meant that a user could spend a given “note” more than once, by keeping the last 32 bits the same but changing some other bits. This would generate different "nullifiers" instead of repeats, which would circumvent the check that should guarantee a note spent only once.

Solution/prevention The Aztec network solved this vulnerability by implementing a bit length check to enforce the used value to be at most 32 bits long. This vulnerability can in general be prevented by ensuring the proper constraining of the circuits used to create and verify the proofs.

4.7 Protocol comparison

In this section we, in this section we summarize the defining characteristics of the zk-SNARK, zk-STARK, and Bulletproof protocols. Table 4.1 shows this comparison. In addition, we briefly describe how we obtained the values listed in that table.

	zk-SNARK [97] [63]	**zk-STARK** [14]	**Bulletproofs** [33]
Proof size	Constant	Polylogarithmic	Logarithmic
Proof generation	Linear	Quasilinear	Linear
Proof verification	Linear	Polylogarithmic	Linear
Trusted setup	Yes	No	No
Quantum secure	No	Assumed	No
Assumptions	(EC)DLP, (B)DHP	Cryptographic hashes	(EC)DLP

TABLE 4.1: Comparison of zk-SNARK, zk-STARK, and Bulletproof protocols

First, for the zk-SNARK, the "Proof size", "Proof generation", and "Proof verification" values were all listed as such by the authors in the introduction chapter of the Pinocchio paper by Parno et al. [97]. Specifically, they state that "key setup and proof generation require cryptographic effort linear in the size of the original computation, and verification requires time linear in the size of the inputs and outputs. Even more surprising, Pinocchio's proof is constant sized, regardless of the computation performed.". The Groth16 SNARK backend has similar asymptotic complexities, as we can examine for arithmetic circuits using Table 2 in the original paper by Groth et al. [63]. Specifically, the proof size is constant, though the constant size depends on the number of group elements configured for the protocol, while the prover complexity is linear in the number wires and multiplication gates in the circuit, and the verifier complexity is linear in the number of elements in the original statement for which the corresponding proof is verified.

Second, for the zk-STARK, Ben-Sasson et al. described the "Proof generation" and "Proof verification" complexities in multiple places in the original paper [14]. The most convenient reference is Figure 2, in which the authors compare their zk-STARK to several other works, where the authors marked the zk-STARK with "Yes" in both the "prover scalability (quasilinear time)" and "verifier scalability (polylogarithmic time)" columns. The zk-STARK protocol proof size complexity was more difficult to observe. In Section 3.2 Theorem 3.4 of the zk-STARK paper [14] the authors state that a Scalable Transparent IOP of Knowledge (STIK) has a polylogarithmic proof length in the number of field elements. They subsequently define STARK to be a realization of a STIK in Section 3.3, which gives it have the same proof size complexity. We affirmed that this polylogarithmic complexity is correct using a page from the StarkWare company, whose founder and CEO of is the main author of the zk-STARK paper, which states that "[STARK] uses cutting-edge cryptography to provide polylogarithmic verification resources and proof size" [112].

Third, for Bulletproof protocol we obtained the "Proof size" from Section 1.1 of the original Bulletproof paper by Bünz et al [33], in which they list their contributions. Specifically, they state that "The proof size is logarithmic in the number of multiplication gates in the arithmetic circuit for verifying a witness.". They list the same in the abstract of the same paper, where they also mention that the proof generation and verification times are linear in the bit length of the range. This complexity applied to range proofs, though. Regarding the R1CS proofs that are more interested in for this work, Section 1.2.3 in the Bulletproof paper [33] describes an arithmetic circuit application, established by their reference to Section 5 in which they explain how arithmetic circuit proofs work in the Bulletproof protocol. In this section, they state that "Constructing the proof and verifying it takes linear time in n.", where n is the number of committed values in each of the two input lists.

We collected the values listed in the "Trusted setup", "Quantum secure", and "Assumptions" rows from other sections in this chapter.

Finally, we remark that each work obtained the complexity of the proof size, proof generation, and proof verification slightly differently, depended on the inner workings of each protocol. Therefore, the complexity may not perfectly correspond between the protocols. We recommend consulting the cited works for more specific size and runtime complexity, and details on how they calculated these complexities.

Chapter 5

Proposed solution

In this chapter, we describe the proposed solution according to the methodology as described in chapter 3. First, in section 5.1 we restate our implementation for the proposed solution, and link this to the research gap observed in our SLR. In section 5.2, we then describe in detail the software and hardware that were used to perform the benchmark, while in section 5.3 we comprehensively describe the implementation of the benchmark design as outlined in section 3.2. After that, we detail the benchmark procedure that we followed to obtain the actual results from our implementation in section 5.4. Finally, we provide a justification for our proposed solution where we briefly state how our proposed solution will address our research questions in this work in section 5.5 and present a schematic overview of our proposed solution in section 5.6.

5.1 Solution

In section 2.2, we previously stated which of the research gaps, observed in our previous SLR, we intend to address in this work. To summarize in a single sentence, we intend to address the lack of a comprehensive applied performance comparison on the three main NIZKP protocols in existing research works. We described our methodology, how we intend to resolve our chosen research gap, in chapter 3. Specifically, in section 3.2 we decided to implement a hash function application using each of the three protocols. Using these equivalent application implementations utilizing several NIZKP protocols, we can benchmark the performance and subsequently compare the resulting metrics between the protocols. To link our implementation back to the observed research gap, by implementing each of the three protocols of interest we provide the comparison between the zk-SNARK, zk-STARK, and Bulletproof protocols that is absent in current literature. We additionally go one step further by implementing these protocols in an equivalent application, which means that we remove the difficulty of comparing the performance between different protocol use cases as was a significant limitation to the protocol comparison in our SLR. By benchmarking each protocol utilized in an identical application, we provide the closest possible comparison between the NIZKP protocols.

5.2 Software & hardware

This section describes our use of software and hardware in implementing and performing the benchmark. Knowing the exact version of each piece of software that we used is important, because different software, and even different software versions of the same software, can induce vastly different implementations which exhibit vastly different performance characteristics. By providing the exact version of each used piece of software, we strive to make our benchmark repeatable by other researchers. Likewise, knowing the hardware used in a benchmark is important because using different hardware can manifest in vastly differing benchmark results. While we would expect different hardware to produce metrics that are proportionate to the speed of the hardware, where the metrics for each protocol change according to the performance of the hardware, this is undoubtedly not guaranteed. Such expectations may particularly not hold when using different processor designs, including different implemented instruction sets (e.g. AVX, AVX2) or an entirely different processor architecture (e.g. ARM instead of x86-64). For this reason, we list the hardware that we used to perform the benchmark, intending to make the benchmark repeatable for other researchers. Alternatively, the list of hardware allows other researchers to explain observed performance differences in reproduced benchmarks when they used different hardware.

5.2.1 Software

For the software, the most important components in the benchmark are of course the ZKP libraries used to implement the three protocols. For this reason, these libraries were the first software that we decided on.

Initially, we started looking at ZKP libraries implemented in the Go language since this was the language with which we were most familiar. It also satisfied our requirement of being a compiled and performant language. We found, however, that only a full-featured zk-SNARK library named Gnark [38] was available in Go. Because of the requirements we set in section 3.1, we should preferably choose a library for each protocol in the same programming language, this would not work. However, we noticed that the Gnark package was well documented and had implemented more primitive building blocks than other libraries we found for the three protocols. For this reason, we found this package interesting to use for initial proof of concept implementations for ideas we thought of. Additionally, we expected that it would be useful to implement our benchmark application in the Gnark package as well, next to the zk-SNARK implementation in the language of the other two protocol libraries. This SNARK implementation in Go could then indicate, when compared to the other SNARK implementation, what potential performance differences a library implementation in a different programming language can make.

This led us to perform a more general cursory search for ZKP libraries, through which we found that Rust had a well-implemented Bulletproof library [125]. We also found and examined several JavaScript libraries, but these did not fulfill our requirement of being written in a compiled and high-performance language. For example, the bulletproof-js library [28] includes a benchmark comparison to other Bulletproof libraries in their documentation, including a comparison to the aforementioned Rust Bulletproof library. This comparison demonstrated that the performance of the bulletproof-js library is several orders of magnitude lower than that of the comparable Rust Bulletproof library, which indicated to us that Rust might be a suitable candidate language to find an implementation for the other ZKP protocols. We also noticed, by not finding any STARK libraries written in either Go or JavaScript, that a full-featured zk-STARK library would be the most difficult to find. Therefore, we focused our attention on finding a good STARK library first. We found a library called libSTARK [10], which is a STARK implementation in C++ by the authors of the original STARK paper. However, our initial impression was that it seemed that this library uses a special notation to design circuits and that we would not be able to freely implement it with the main programming language. We furthermore found the Rust Winterfell crate [49], which seemed well-implemented, provided documentation, and was in active development. There were some limitations to this library though, including that it does not implement perfect zero-knowledge and focuses on succinctly proving computations instead of knowledge. We will describe these limitations in more detail in section 5.3. However, even with these limitations in mind, it was the best option we found. We already identified the Rust Bulletproof crate earlier, which meant that we only had to find a SNARK library to have discovered a library for each protocol in the Rust language. We found this in the Rust Bellman crate [124]. With us unearthing a full-featured library implementation for all three protocols written in Rust, we decided to implement our benchmark in Rust. Besides having a library implementing each protocol, the libraries were each well-implemented, at least somewhat documented, and well-known. In summary, we found that implementing the ZKP application in Rust using the Bellman, Bulletproof, and Winterfell crates was the best option for our benchmark.

To summarize, we ended up using four ZKP libraries written in two different programming languages. Since our benchmark implementation depended on these ZKP protocol libraries, we included those as our main dependencies. We additionally depended on several cryptographic libraries required for using the mentioned NIZKP libraries. We detail the full list of (direct) dependencies by language in Table 5.1.

Language	Dependency name	Dependency version
Go	github.com/consensys/gnark	v0.9.1
Go	github.com/consensys/gnark-crypto	v0.12.2-0.20231013160410-1f65e75b6dfb
Rust	bellman	0.14.0
Rust	bls12_381	0.8.0
Rust	bulletproofs	4.0.0 (with ‘features = ["yoloproofs"]‘)
Rust	curve25519-dalek-ng	4.1.1
Rust	ff	0.13.0
Rust	merlin	3.0.0
Rust	rand	0.8.5
Rust	winterfell	0.8.1
Rust	blake3	1.5.1 (with ‘default-features = false‘)
Rust	criterion	0.5.1 (with ‘features = ["html_reports"]‘)

TABLE 5.1: Programming dependencies used to implement the benchmark

Because of our chosen ZKP libraries, we required the usage of the two programming languages Go and Rust, as well as the Rust package manager Cargo. The used version for each software is listed in Table 5.2.

Name	Version
Go	1.22.0
Rust	1.76.0
Cargo	1.76.0

TABLE 5.2: Software used to implement the benchmark

5.2.2 Hardware

As for the used hardware, we performed the benchmarks on a desktop computer with the following specifications:

- AMD Ryzen 9 5900x processor

- 32GB DDR4 3600MHz memory (2x 16GB in dual channel)

The computer ran Windows 10 version 22H2 as the operating system and we configured it to run in the better performance power mode. The D.O.C.P. (Direct Overclock Profile) setting was enabled in the motherboard settings to attain the intended speeds as specified for the memory modules. We did not apply any further overclock or undervolt, meaning that the processor ran at stock speeds.

5.3 Implementation

Now that we determined which software and dependencies we want to use to implement the benchmark, we describe the actual implementation of the benchmark using the chosen ZKP libraries.

Our initial idea for the implementation, as described in section 3.2, comprised of a zero-knowledge proof which proved that a given public Elliptic Curve Digital Signature Algorithm (ECDSA) key verified a signature and is included on a list of trusted keys. The intention for such proof was to prove that the user utilized a hardware security key from a trusted manufacturer to sign a message, without leaking the manufacturer details or batch information of the hardware security key. Our benchmark application would have implemented such proof for each of the three ZKP protocols, albeit without communicating to a real hardware security key, generating the public keys in code instead. Our first step in creating the implementation was to create a proof of concept using the Gnark zk-SNARK library. We chose to implement the proof of concept in Gnark because of the great documentation, familiarity with the language, and numerous existing cryptographic primitives that the codebase contained. We started out with an implementation using the Edwards-curve Digital Signature Algorithm (EdDSA) to get familiar with the Gnark library since creating a Gnark circuit for proving the verification of an EdDSA signature was explained in a tutorial [47]. We expanded this proof to additionally verify that the used public key was included in a provided list of trusted public keys. We defined the public key as a secret input to the circuit, while we set the message, signature, and trusted key list as public inputs. The code for this implementation can be found in the Git repository for this research [94]. With a working implementation for EdDSA, we re-implemented the same approach in Gnark for ECDSA. This process was more involved, because we had to use more primitive cryptographic building blocks, yet eventually we got the ECDSA-proof circuit working identically to the EdDSA circuit. We should note though that, since we ended up not using this implementation, we did not fully implement some aspects of the proof that did not impact functionality but would have impacted security in any real use cases. The corresponding code can be found in our Git repository [94].

Now that we had a working zk-SNARK implementation using the Gnark library, we knew that the idea would technically be possible to implement. With that said, we did have to implement the same application for each of the three ZKP protocol libraries in Rust, which is where we hit some difficulties. First, while we implemented the proof-of-concept idea in Gnark because it provided a tutorial, documentation, and many cryptographic primitives, this was not the case for the Rust ZKP libraries. This meant that we would have had to implement these primitives ourselves, leading to more opportunities for security issues. More importantly, we expected that this would take more time than we had available for the research. Even more critically, their creators geared the zk-STARK library towards succinctly proving computations, as opposed to knowledge like the zk-SNARK and bulletproof libraries. This meant that the application would require a completely different approach in the STARK implementation compared to the other two protocols. On top of this, at the time of implementation, the STARK library did not provide perfect zero-knowledge. This meant that there was no option for us to provide the used public key to the circuit, as required in our proof of concept since the proof would not keep this key private. While it sounds strange to have to keep a public key secret, we reiterate that openly providing this key would reveal some privacy-sensitive information about the used hardware security key. As a result, doing so would invalidate the entire reason for utilizing a NIZKP in the application in the first place. For these reasons, we decided to abandon this idea for our benchmark application. Instead, we opted to use a more rudimentary application.

For the basic ZKP application idea that we could implement more equally for all three protocols, we decided to implement a hash function. Our application would ensure this hash either had a variable number of rounds or would use the hash as part of a hash chain, to enable some way to increase the required amount of work in the proof. After some deliberation between the MiMC [2], Poseidon [62], and Rescue [114] hashes, we eventually chose the MiMC hash function. Namely, this hash function is well-optimized for zero-knowledge proofs [11], has a simple algorithm that is easy to implement in proof circuits, and example implementations we could adapt and build on were available for the SNARK and Bulletproof Rust ZKP libraries. The number of rounds used in the MiMC hash can be varied in our benchmark, where each round requires a different round constant for security. This enabled us to implement the hash for all three protocols, since, at least for our intents and purposes, proving knowledge of the pre-image of a public hash is the same as proving the computation of calculating the required hash from a pre-image provided by the prover. Though, in the latter case, applicable to the STARK implementation, the pre-image would not necessarily remain private. For equality reasons, we therefore did not focus on these variables remaining private in the other protocols either. This is a limitation of our benchmark, for which we decided that the most important aim was to keep the proof as similar as possible. Since this limitation is important to consider for real-world implementations using ZKPs, we further discuss this limitation in chapter 7 section 7.4.

To summarize, our actual implementation existed of a proof that verifies that the prover knows a pre-image to a certain MiMC hash image. The MiMC hash had a variable number of rounds, and we provided the round constants as input to the circuit. We implemented this application in each of the three chosen Rust protocol libraries. Our implementation adapted and built upon example implementations for both the Rust SNARK library [8] and Bulletproof library [30], while we created the Winterfell STARK library implementation from scratch. Moreover, we implemented the application in the Go Gnark zk-SNARK library as well, for comparison reasons described in section 5.2. We conjecture that this implementation provided the best possible comparison between the three protocols. Where significant for such real-world implementations, we provide additional protocol-specific context in chapter 6 and chapter 7. We also present additional justification for our implementation idea in section 3.2. The code for all implementations can be found in the Git repository for this research [94].

An important consideration for the Bulletproof implementation was that we did not apply any form of batch verification, even though this is one of the beneficial aspects of the Bulletproof protocol that the Bulletproof library implements. While such batching verification could reduce the total verification time compared to performing each proof verification separately, it required an application where such batching is viable. In this work, we benchmarked the process of generating and verifying a single proof, which means that batching did not apply to our benchmark. We will discuss the implications of this in chapter 7.

Finally, when inspecting our implementation, one should consider that we used seeded randomness for our benchmark. This means that the randomness we used in our implementation is not secure. Any real-world implementation should at minimum replace the seeded randomness with a cryptographically secure randomness source.

5.4 Benchmark procedure

With the implementation code completely written, we commenced the benchmark procedure. First, we restarted the hardware which we performed the benchmark on to clear as many resources as possible. After this restart we waited a minute for the operating system and all initiated startup processes to settle. We then opened a separate terminal window in the Rust and Go implementation directories.

The first benchmark we performed was the benchmark comparing the protocols on several numbers of rounds. For the number of rounds, we settled on the numbers corresponding to $2^x - 1$ with $x \in \{4, 6, 8, 10, 12\}$, since this formula is a requirement for the zk-STARK implementation as described in section 5.3. This gave us the set of MiMC rounds $\{15, 63, 255, 1023, 4095\}$, which we believe provided a nice range to represent the performance differences between the NIZKP protocols for various amounts of required work. We ensured that we applied the correct default configurations and had set the desired number of MiMC rounds in the benchmark code. We then issued the ‘cargo bench‘ command, which compiled the Rust code as a release target for the best performance and used this compiled binary to run the benchmark for each of the three protocols sequentially. When the benchmark for the Rust implementations was complete, we logged the benchmark results and other metric outputs in an Excel sheet for each protocol under the set number of MiMC rounds. With the Rust benchmark results recorded, we switched to the other terminal for the Go implementation and repeated the process, only using the ‘go test -bench . ./internal/hash/.‘ command instead. This command, like the ‘cargo bench‘ command for Rust, compiled the Go SNARK MiMC implementation and ran the benchmark outputting the results. When we performed all benchmarks for a given number of MiMC rounds, we repeated the process for each other number of rounds, noting down all the results in the same Excel sheet.

We additionally ran a benchmark comparing the performance of the zk-STARK implementation for different options. The process for this benchmark resembled the procedure described above, yet instead of using fixed option parameters with a dynamic number of rounds, we fixed the number of rounds and modified the default option parameters by a single option at a time. By initiating the ‘cargo bench stark‘ command, we conducted the benchmark for just the zk-STARK implementation and obtained the performance difference caused by a single option parameter change. We then recorded the benchmark results and metrics in the Excel sheet and subsequently reverted the option parameter to the default, repeating this process for all options and several parameters for each option. Finally, we performed one final benchmark for the STARK, in which we set the option parameters to a combination of values that provided the best performance according to the individual parameter benchmarks.

Now that we performed all benchmarks, we processed the metrics in the Excel sheet into the benchmark result tables and graphs found in chapter 6 section 6.1. The code that we wrote to implement all benchmarks can be found in the Git repository corresponding to this work [94].

5.5 Justification

Now that we depicted our proposed solution in-depth, we succinctly provide a justification for how this proposed solution addresses the research questions as stated in section 1.2. We address the first research question, "What are the performance differences between the three included NIZKP protocols, as observed from a real-world implementation of each protocol in an application that is as equal as possible, expressed in efficiency and security level?", with our proposed solution. By implementing the identical MiMC hash application utilizing a real-world library implementation for each of the three included NIZKP protocols, we will be able to observe the performance metrics related to the efficiency and security level for each. While the performance and security metrics available in each protocol will limit our scope, we can compare the metrics that we were able to obtain for each protocol to provide an answer to this first research question. By extracting the strengths of each included NIZKP protocol from the performance metrics, and cross-referencing these with the unique requirements of several applications, we can distil knowledge on the use case contexts that are most beneficial for each protocol. Using this extracted knowledge, we will then be able to answer the second research question, which should provide researchers with recommendations on the situations in which a given NIZKP protocol is best applied. To conclude, we express our confidence that by implementing the proposed application we will be able to provide a comprehensive answer the research questions stated at the start of this work. We consider this to constitute sufficient justification to implement our proposed solution.

5.6 Overview

To conclude this chapter, we provide a schematic overview of our proposed solution in Figure 5.1.

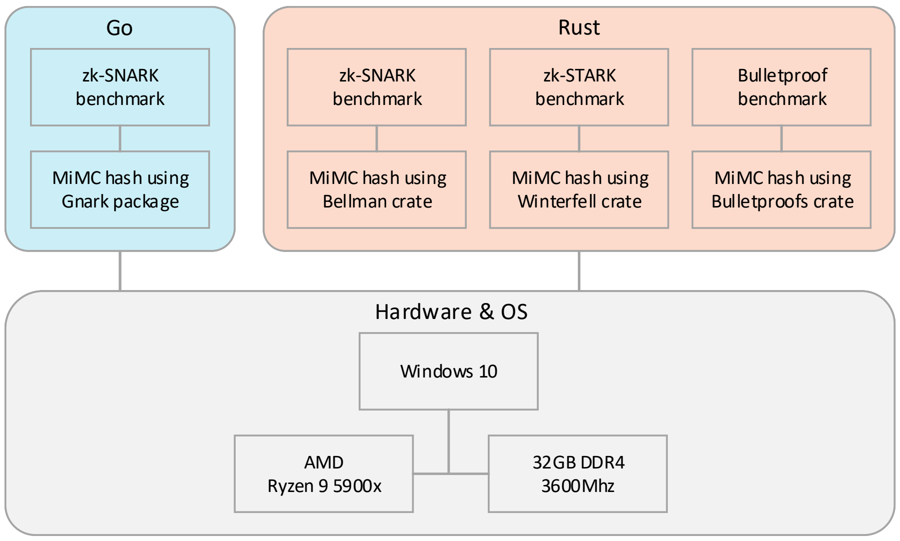


FIGURE 5.1: Schematic overview of our proposed solution.

Chapter 6

Results

In this chapter, we detail and analyse the findings collected from our benchmark. In section 6.1, we list the benchmark results in the form of tables, with some explanations and complementary context for the metrics. In addition, we provide graphs as an alternative way to compare the performance differences between the ZKP protocols. Subsequently, we analyse the raw benchmark data and provide more context on the data in section 6.2. In this analysis, we dive deeper into the differences between the ZKP protocols and any anomalous results we obtained from our benchmark.

6.1 Benchmark results

In this section, we report the results from the benchmark which we implemented as described in section 5.3 and subsequently performed according to the procedure described in section 5.4. Before listing the results, however, we first provide some context on the abbreviations used to list the results, next to the configuration we used for each protocol.

6.1.1 Abbreviations

Within Table 6.1, Table 6.2, and Table 6.3, the following abbreviations are used to save space, which enabled us to fit the tables on a single page:

- **Rnds** - Rounds; The number of rounds used in the MiMC hash.
- **Protocol** - The NIZKP protocol and corresponding programming library.
  - **Bulletproof** - Used the Rust Bulletproofs crate v4.0.0 [29] [125].
  - **SNARK (R)** - Used the Rust Bellman crate v0.14.0 [7] [124].
  - **SNARK (G)** - Used the Go Gnark package v0.9.1 [58] [38].
  - **STARK** - Used the Rust Winterfell crate v0.8.1 [121] [49].
- **CRS (B)** - Common Reference String; The size of the CRS (without verification key) in bytes.
- **VK (B)** - Verification Key; The size of the verification key in bytes.
- **W (B)** - Witness; The size of the full witness in bytes.
- **PW (B)** - Public Witness; The size of the public witness part in bytes.

- **C (B)** - Commitments; The size of the commitments in bytes.
- **P (B)** - Proof; The size of the proof in bytes.
- **CT (ms)** - Compile Time; The time required to compile the circuit in milliseconds.
- **ST (ms)** - Setup Time; The time required to perform the setup in milliseconds.
- **PT (ms)** - Proof Time; The time required to generate the proof in milliseconds.
- **VT (ms)** - Verification Time; The time required to verify the proof in milliseconds.
- **SC (b)** - Security Conjectured; The conjectured security level in bits.
- **SP (b)** - Security Proven; The proven security level in bits.
- **Option** - The option for which the parameter was changed from the default. If (D) is appended to one of the option names, then this parameter is our chosen default.
  - **NQ** - NUM_QUERIES; The number of queries performed to verify correctness.
  - **BF** - BLOWUP_FACTOR; The factor that determined the probability of detecting a false proof in each query.
  - **GF** - GRINDING_FACTOR; The factor that impacted the security of the proof by requiring a certain number of leading zeros in specific hashes, resembling a proof-of-work.
  - **FFF** - FRI_FOLDING_FACTOR; The factor by which each iterative round reduced the degree of the polynomial.
  - **FRMD** - FRI_REMAINDER_MAX_DEGREE; The maximum degree of the remainder polynomial.
  - **Hash** - Hasher; The algorithm we set to calculate hashes within the protocol.
  - **FE** - FIELD_EXTENSION; Field extensions enabled higher proof security than possible with just the finite field.

6.1.2 Configurations

For the main benchmarks, we chose a default configuration for each of the three protocols. In the Bulletproof protocol implementation, there were not a lot of configuration options. The protocol implementation depended on the curve25519_dalek_ng crate [41], which means that the protocol used the Curve25519 elliptic curve in combination with the Ristretto group [104]. This group enabled the construction of prime-order elliptic curve groups that had the special property of a non-malleable encoding. Furthermore, the Bulletproof protocol implementation depended on the Merlin crate [82], which implements proof transcripts and automated the Fiat-Shamir transform [51]. Besides the dependencies, we used the following configuration for the Bulletproof implementation:

- Bulletproof generators capacity: This number had to be larger than the number of multipliers in the circuit, rounded to the next power of two. We accordingly set the Bulletproof generators capacity to $(m + 1) * 2$, where m is the set number of MiMC rounds.

- Pedersen commitment generators: We used the default option provided by the library, meaning that we configured the usage of the ristretto255 base point and SHA3-512 hash of the same base point for the blinding.

The zk-SNARK implementation libraries, similarly, did not provide a wide range of configuration options. We configured both the Rust and Go implementations to use the BLS12-381 pairing-friendly elliptic curve [25] for the scalar field and pairings. For the Go code, we used the BLS12-381 implementation in the gnark-crypto package [39], while we used the bls12_381 crate [21] for the Rust code. Additionally, both implementations used the Groth16 [63] proof system to implement the zk-SNARK proof, a system that both protocol libraries had built in. We did not select any further configuration parameters. Where required, we generated any other parameter randomly.

Finally, the zk-STARK library provided the most options for the configuration of all protocols and implementing libraries. Considering that the STARK implementation did not have any dependencies outside of the Winterfell crate itself, we only had to choose the default STARK configuration parameters:

- Number of Queries (NQ): 42
- Blowup Factor (BF): 8
- Grinding Factor (GF): 16
- FRI Folding Factor (FFF): 8
- FRI Remainder Maximum Degree (FRMD): 31
- Hasher (Hash): Blake3_256
- Field Extension (FE): None

We explain the meaning of these configuration options in subsection 6.1.1. We chose these configuration parameters because they provided a good security level and were reasonable options near in the middle of possible configurations in most cases. However, as described in section 5.4, we also performed a benchmark for different configuration parameters for the zk-STARK protocol. This further compared the performance difference that the configuration parameters can make since configuration options were numerous enough that using just one configuration could have displayed a distorted view of the protocol performance. The results of the configuration parameter benchmark can be found in subsection 6.1.3.

6.1.3 Results

Now that we described the abbreviations and configurations used for the benchmarks, we can start listing the benchmark results.

The results from the benchmark for each protocol, using the default configuration as described in subsection 6.1.2, can be found in Table 6.1 and Table 6.2. Table 6.1 lists the sizes in bytes of different data, provided as inputs and outputs. As one can observe, the proof size was the only metric available for all three protocols and all four implementations. The CRS, because of the trusted setup requirement that is only applicable to the SNARK protocol, was only available for the two SNARK implementations. Similarly, the witness was only available for the Go zk-SNARK implementation because that library generated the witness in a separate step. After creation, the library used the witness as input to the proof-generating function, next to the proving key and the constraint circuit. The proof-generating function in the Rust implementation, on the other hand, only accepted the circuit and CRS as input. The library presumably generated the witness internally, which we could therefore not directly measure in our benchmark. Lastly, the commitment size was only available in the Bulletproof protocol yet served a similar purpose to the witness in the SNARK protocol.

Table 6.2 lists the proof generation and verification times, in milliseconds, next to the security level in bits. In this table as well, we only list the results that we could obtain from each protocol implementation. As shown, only the proving time and verification time metrics were available for all three protocols and all four implementations. Just like for the size benchmarks, the setup time metric corresponding to the trusted setup was only available for protocols that require a trusted setup, meaning just the two zk-SNARK implementations. The compile time, only available to the Go SNARK implementation, was a separate step in the Go SNARK implementation. For this reason, we recorded it separately. The Rust SNARK library was written such that other steps include the compile time; the compilation is not a separate step. Since at one point the circuit had to be transformed in a constraint system, and unlike in the Go implementation the Rust implementation took the uncompiled circuit as input to the proof-generating function, we expect the burden of the compile time from the Go implementation was included in the proving time for the Rust implementation. We consider this in our analysis in section 6.2 and discussion in chapter 7.

Finally, the conjectured and proven security levels of the proof in bits were only available from the protocol in the STARK implementation. The other protocols, sadly, did not implement any functionality to obtain the security of the proof as configured. While we know from chapter 4 which cryptographic assumptions are made for each protocol, and that only the zk-STARK protocol is considered quantum resistant, since the security of the proof is dependent on which cryptographic protocols were used underneath, the proof circuit, and for example also the security of the input, this does not explain the exact security level of each proof that we created. Rectifying this limitation, while possible, would require an extraordinary amount of time, theoretical protocol knowledge, and knowledge of the practical library implementations. We therefore consider this to be outside of the scope of this research work and will elaborate on this limitation in section 7.4. While this also means that we were unable to provide a full picture, we will make a best effort to provide a security level comparison regardless in section 6.2 by collecting security level metrics from works by other researchers. For theoretical security comparisons, we refer the reader to chapter 4.

We then continued by performing the configuration benchmark for the zk-STARK protocol implementation, in which we changed a single configuration parameter at a time to measure the performance impact. Table 6.3 lists the performance metrics obtained from that benchmark for the metrics available to the STARK implementation. The first column, "Option", denotes the configuration parameter that we changed the default value of. We grouped the options by different values for the same parameter and marked the default parameter with (D). There are a few things to note in this table. First, the GF 32 benchmark does not have a listed result. This is due to the benchmark for this parameter not finishing a single iteration after a few minutes. Second, the FE Cubic benchmark, equally, does not have any results. This absence came as the result of the library not implementing the cubic field extension for our use, as specified by the library in a returned error.

Finally, with the results for the zk-STARK implementation configuration benchmark in hand, we wondered what would happen if we combined all the best performing parameters together. Would the performance differ significantly from our configured default? To investigate this, we configured the zk-STARK implementation with the following 'best' parameters, where we made sure the conjured security level would not go below 100 bits:

Rnds	Protocol	CRS (B)	VK (B)	W (B)	PW (B)	C (B)	P (B)
15	**Bulletproof**	-	-	-	-	64	737
15	**SNARK (R)**	6816	528	-	-	-	192
15	**SNARK (G)**	10538	1448	588	524	-	484
15	**STARK**	-	-	-	-	-	6657
63	**Bulletproof**	-	-	-	-	64	865
63	**SNARK (R)**	27552	528	-	-	-	192
63	**SNARK (G)**	40778	3752	2124	2060	-	484
63	**STARK**	-	-	-	-	-	16518
255	**Bulletproof**	-	-	-	-	64	993
255	**SNARK (R)**	110496	528	-	-	-	192
255	**SNARK (G)**	161738	12968	8268	8204	-	484
255	**STARK**	-	-	-	-	-	24866
1023	**Bulletproof**	-	-	-	-	64	1121
1023	**SNARK (R)**	442272	528	-	-	-	192
1023	**SNARK (G)**	744562	49832	32844	32780	-	484
1023	**STARK**	-	-	-	-	-	38769
4095	**Bulletproof**	-	-	-	-	64	1249
4095	**SNARK (R)**	1769376	528	-	-	-	192
4095	**SNARK (G)**	2978234	197288	131148	131084	-	484
4095	**STARK**	-	-	-	-	-	55132

Table 6.1: Size results of the protocols benchmark.

- Number of queries: 41; lower tested numbers showed better performance, at least for proof size and verification time, but reduced the security level below our set threshold.
- Blowup factor: 16; slightly increased the proof size and verification time, but strongly reduced the proof generation time. Blowup factors of 8 or lower demonstrated even better performance, yet they reduced the security level to a value below our set threshold.
- Grinding factor: 8; had the best proof size, a proof time equivalent to lower values, and a proof verification time equivalent to grinding factor 24.
- FRI folding factor: 4; showed the best proof and verification time metrics, while the proof size was only slightly larger than for the default FRI folding factor of 8.
- FRI remainder maximum degree: 255; the highest possible maximum remainder degree for the FRI had the best performance in all three metrics of proof size, proof time, and verification time, while not appearing to have impacted the security level.

We changed neither the hasher nor the field extension from the default. The Blake3_192 hasher, as expected, showed better performance than the Blake3_256 hasher for proof size and time, with a similar verification time. The quadratic field extension, while almost halving the proof time, significantly increased the proof size and verification time. Besides displaying worse metrics, we worried that a different field extension would have an impact that would make it hard to compare the performance of the optimized parameters against the performance of the default values. We therefore did not alter this setting. We note that, while in most cases the conjured security level remained the same or at least

Rnds	Protocol	CT (ms)	ST (ms)	PT (ms)	VT (ms)	SC (b)	SP (b)
15	**Bulletproof**	-	-	6.756	0.899	-	-
15	**SNARK (R)**	-	10.467	4.479	1.703	-	-
15	**SNARK (G)**	0.043	3.425	1.299	1.138	-	-
15	**STARK**	-	-	2.060	0.052	120	73
63	**Bulletproof**	-	-	25.210	2.677	-	-
63	**SNARK (R)**	-	18.643	5.563	1.686	-	-
63	**SNARK (G)**	0.227	10.292	2.420	1.195	-	-
63	**STARK**	-	-	0.552	0.142	118	75
255	**Bulletproof**	-	-	102.450	11.069	-	-
255	**SNARK (R)**	-	42.788	12.218	1.709	-	-
255	**SNARK (G)**	1.830	40.888	5.676	1.407	-	-
255	**STARK**	-	-	11.339	0.199	116	74
1023	**Bulletproof**	-	-	499.610	92.663	-	-
1023	**SNARK (R)**	-	132.280	30.268	1.684	-	-
1023	**SNARK (G)**	10.453	150.211	19.867	2.280	-	-
1023	**STARK**	-	-	13.094	0.313	114	73
4095	**Bulletproof**	-	-	3614.500	1271.200	-	-
4095	**SNARK (R)**	-	440.560	96.865	1.695	-	-
4095	**SNARK (G)**	42.937	453.436	61.512	5.733	-	-
4095	**STARK**	-	-	44.876	0.452	112	72

TABLE 6.2: Time and security level results of the protocols benchmark.

above our stated threshold of 100 bits of security, the proven security level was usually affected negatively when choosing more performant configuration parameter values. When configured with the stated optimized parameters, we obtained the metrics as shown in Table 6.4.

6.2 Analysis

Now that we have detailed all the obtained benchmark results, we start with our analysis of those results.

First, we analysed the differences between the Bulletproof, zk-SNARK, and zk-STARK protocols. To this end, we created some additional graphs that show the obtained metrics as a plot for each protocol, which also shows the change in this metric for different numbers of MiMC rounds. Figure 6.1 shows the size of the proof generated by each protocol implementation and the difference that an increasing number of MiMC rounds makes for this metric. Figure 6.2 and Figure 6.3 show a similar plot for the proof generation time and proof verification time metrics, respectively.

As one can see from the metrics in Table 6.1 and the plot in Figure 6.1, there is a clear distinction between the proof sizes in the four implementations. The SNARK protocol implementations had the smallest proofs, with a size of 192 bytes for the Rust implementation and 484 bytes for the Go implementation. The proof size was also constant for both, meaning that the size of the proof remained the same, independent of the number of MiMC rounds. This was different for the Bulletproof and zk-STARK implementations, which both displayed a proof size that increased with the number of MiMC rounds. The proof size of the STARK protocol was larger than that of the Bulletproof protocol and additionally grew more rapidly in size with the number of MiMC rounds than the Bulletproof proof.

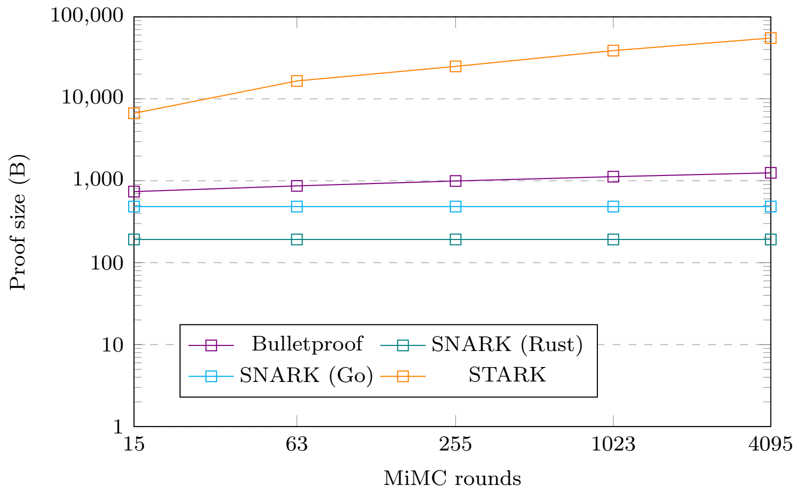


FIGURE 6.1: Proof size benchmark plot

This observation, however, fails to capture the broader perspective of data that needs to be transferred. The two SNARK protocol implementations may have had the lowest proof sizes, they additionally required the verifier to obtain the verification key. This key was a constant additional 528 bytes for the Rust implementation, or an increasing size starting at 1448 bytes for the Go implementation. For us to obtain the total data size as required by the verifier, we summed these figures. This resulted in the data size from the Rust SNARK implementation, a total of 720 bytes, suddenly being just shy of the Bulletproof implementation data size. Having said that, the size of the Rust SNARK implementation was nonetheless still constant, whereas the data size for the Bulletproof implementation grew with the number of hash rounds. At the same time, the combined data size of the Go SNARK implementation grew even faster in the number of MiMC rounds. Besides, the combined amount of data was already larger than for the Bulletproof, even without the public witness the verifier required to verify a proof in this implementation. By 1023 MiMC rounds, the amount of data from the combined verification key and proof size in the Go SNARK implementation was higher than for the STARK implementation. This showed a clear contrast between the two zk-SNARK implementations, an aspect which we will deliberate on in chapter 7.

After the proof size, we now examine the proof generation times, as detailed in Table 6.2 and plotted in Figure 6.2. As one can see, the Bulletproof protocol implementation demonstrated the slowest proof-generating time, followed from a distance by the two SNARK implementations. Additionally, even though all protocol implementations showed the proof generation times to be increasing with the number of MiMC rounds, the Bulletproof implementation proving time increased faster than the other three implementations. The two SNARK implementations performed similarly in this metric, and performance between the two converged at higher numbers of MiMC rounds. Especially at lower round numbers, however, the Go implementation performed better than the Rust implementation. Having said that, the Go SNARK implementation required a separate compile time, which the Rust implementation did not need. For lower numbers of MiMC rounds, this compile time was negligible, yet towards higher round numbers this compile time grew and became significant. So significant, in fact, that when added to the proof generating time, the Go implementation converged with the Rust implementation at 1023 rounds. For any larger number of rounds, the combined compile and proving time in the Go library

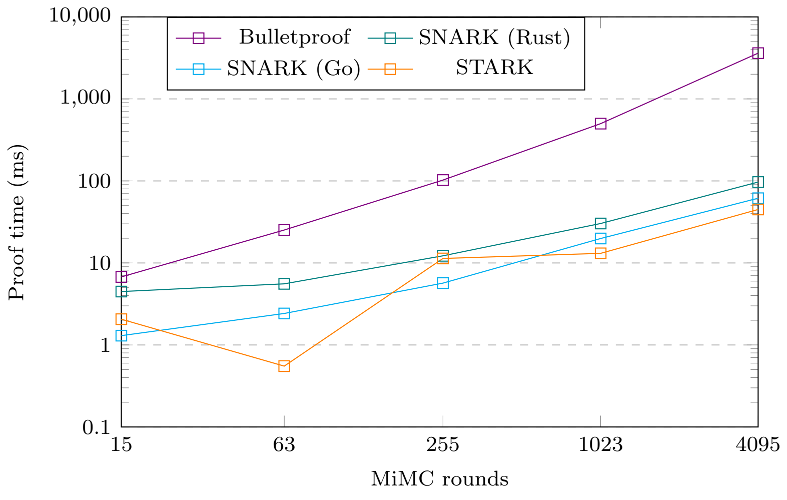


FIGURE 6.2: Proof time benchmark plot

demonstrated a higher combined compile and proving time than the Rust library. The zk-STARK implementation's proof time metrics showed some intriguing fluctuations. These fluctuations made it beat the Go SNARK implementation for some numbers of MiMC rounds while losing out to it in others. Especially the 63 MiMC rounds benchmark metric is perplexing since the proof generating time was much faster than at 15 MiMC rounds. At first, we suspected this result to be a fluke in our benchmark. Re-running the same benchmark multiple times, however, provided us with consistent results throughout each attempt. This indicated that the performance fluctuation was caused by something other than a problem in our benchmark. We therefore attribute the performance fluctuation to some number internal to the protocol, related to the number of MiMC rounds, being optimal for the FRI process at 63 MiMC rounds, especially compared to the same number for the 15 rounds benchmark. We elaborate on this topic in our discussion in chapter 7. In general, the data and graphs displayed that the zk-STARK and two zk-SNARK implementations had a proof time in the same order of magnitude, while the Bulletproof protocol was slower in generating proofs. Next to that, the proof time increased more rapidly with the number of rounds for the Bulletproof implementation than in other implementations.

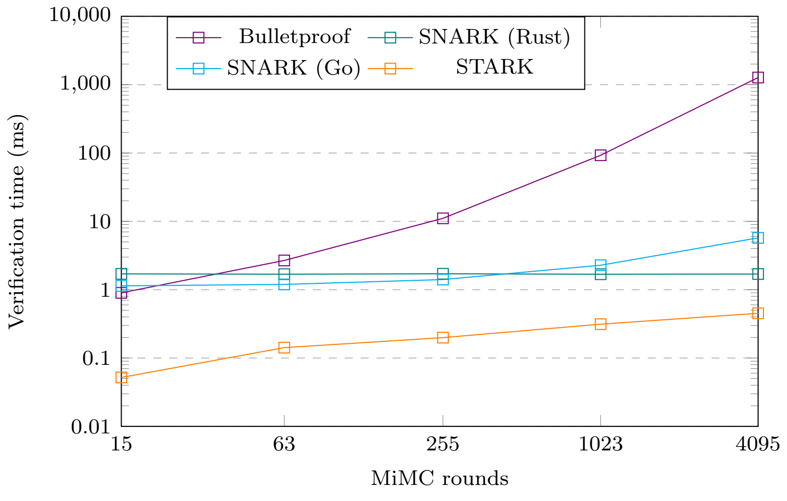


FIGURE 6.3: Verification time benchmark plot

We now change our focus from the proof generation times to the proof verification times, which we plotted in Figure 6.3 from the data in Table 6.2. Our first observation is that the rankings between the protocols were like those for the proof generation times. The Bulletproof protocols showed the slowest proof generation times, whereas the two zk-SNARK implementations demonstrated a comparable proof verification time. The zk-STARK implementation demonstrated the fastest proof verification times throughout. Upon closer inspection, though, there are several more differences. First, the Bulletproof implementation temporarily had a faster proof verification time than the two STARK implementations for the lowest number of benchmarked MiMC rounds. Second, unlike the Go SNARK implementation, which showed slightly increasing verification times for larger numbers of MiMC rounds, the Rust implementation verification times were constant within the margin of expected variability of a benchmark. As for the proof generation times, this means that the Rust implementation became faster than the Go implementation at higher numbers of MiMC rounds. Third, especially at low round numbers, the zk-STARK protocol was around an order of magnitude faster than the two zk-SNARK implementations. Given that the verification times for the STARK increased though, while those of the Rust STARK implementation remained constant, it is conceivable that the STARK implementation would have lost this advantage for even larger numbers of MiMC rounds. This observation involves us extrapolating the data though, it is not something we can conclude from our benchmark data.

The final analysis for the comparing benchmark is the security level of each protocol. As specified in subsection 6.1.3 and reflected in Table 6.2, we could only obtain the conjured and proven security level in bits from a function in the zk-STARK implementation library. This made it hard to directly compare the security level for each implementation, which we will indicate as a limitation in section 7.4. However, we could obtain an expected security level for the protocol implementations from referential works by others. A work by Aranha et al. [4] surveyed several elliptic curves for proof systems, including the BLS12-381 curve. They specified the BLS12-381 curve, the curve used in both our SNARK implementations, to have a 127- or 126-bit security for the group and prime field, respectively. While they likewise discussed curve25519 as used in the Bulletproof implementation, they did not mention any security level. Because the only configuration option for the zk-SNARK implementation was the used elliptic curve, as discussed in subsection 6.1.2, we assume that the curve alone decided most of the protocol security in the SNARK implementation. This would give the two SNARK implementations the same almost 128-bit level security as stated for the BLS12-381 curve, which we expect to be a conjured security level and not a proven one. Similarly, because the Bulletproofs paper [33] only mentioned the security of the protocol in the context of the used libsecp256k1 curve, we expect the curve to define the burden of the security level of the protocol. Since our Bulletproof protocol benchmark implementation used Curve25519, which provides an approximately 128-bit security level [91], we hypothesize this to be the conjured security level of the Bulletproof implementation as well. This is not the case for the zk-STARK, for which Ben-Sasson et al. Described the proven security bound in their work [13]. As they demonstrated, the conjured security level for zk-STARK is the minimum between a number calculated from the number of queries and grinding factor, the collision resistance of the used hash, and a number calculated from the field extension and trace length [48]. The lack of direct numbers for the security level of each protocol implementation in our benchmark resulted in uncertainty, though from the hypothesized numbers that we obtained from a spectrum of sources, the best we could infer was that the security level for the three protocols feature a comparable conjured security level. Yet, for this conclusion, we admittedly did not consider several practical factors in the SNARK and Bulletproof protocols. For this reason, we state that the conclusion does not provide a comprehensive view.

At last, we analysed the benchmark comparing the different configuration parameter values in the zk-STARK protocol implementation. First, we dissected the obtained metrics for changing each configuration parameter, starting with the number of queries. As can be seen in subsection 6.1.2, the proof size and verification time increased with the number of queries. This makes sense since the more queries, i.e. checks in the protocol, the protocol had to perform, the more work had to be included in the proof and verified. This can be observed clearly in the results, in that the number of queries determined a large part of the security level. The one metric that behaved anomalously to the expectation in this regard was the proof time metric. Even when the prover did not have to perform any additional work for a larger number of queries, this does not explain why the benchmark results drastically differ between even small value changes. Furthermore, these metrics neither consistently go up or down, which is explicitly visible when looking at the sixfold increase in the proof time between 41 and 42 queries. We currently do not have an explanation for this phenomenon, yet the results for this metric were intriguing. Next up is the blowup factor. For this parameter, we could see a clear increase in the proof size and verification time. Besides some fluctuation, the proof time also seemed to increase with a larger blowup factor, especially towards higher values. This observation can be accounted for by an increasing blowup factor leading to a higher likelihood that a verifier detects a false proof. In turn, this can be observed in the security level increasing with the blowup factor and the additional work that this required. We now look at the grinding factor, which determined a specific number of leading zeros in hashes, resembling a proof-of-work like concept. This would require extra work from the prover for larger grinding factor values, which is indeed what we observed. In return for this extra work, the proof demonstrated a higher proven security level, though the conjured security level remained identical. The verification time, furthermore, did not significantly shift outside of the variation expected from a benchmark. The proof size, on the other hand, fluctuated in a manner that we cannot explain with benchmark variation. Instead, the small variation of a few thousand bytes indicated an expected proof size difference, initiated by fluctuations in parameters internal to the protocol that the proof had to include. The FRI folding factor did not show a clear increase or decrease in the proof size, proof time, and verification time metrics with the size of the parameter value. Instead, it seems that the optimum balance was somewhere in the middle. Whereas a folding factor set to 8 provided an optimal proof size, a value of 4 provided optimal proof generation and verification times. These optimum values were consistent with the impact that the FRI folding factor had, namely that it determined how much each iterative round reduced the degree of the polynomial. Large values would therefore mean that each iterative round had to reduce the polynomial degree by a large amount, requiring a lot of work. Small FRI folding factor values, on the contrary, would require a lot of iterations to reduce the polynomial to the desired degree. The FRI folding factor did not seem to influence the security level. Lastly, there was the FRI remainder maximum degree parameter, an increase that generally led to a smaller proof size and lower proof verification time. The proof time overall showed the same trend, though as it did for the number of queries and the blowup factor, it fluctuated significantly. The observation that the proof size and verification times went down with a higher maximum remainder degree makes sense given that this value allowed a polynomial to have a higher maximum remainder degree. This enabled the protocol to not reduce the degree of the polynomial as much, which removed the need for the proof to include these additional iterations. This furthermore reduced the work required for the verification. From our benchmark results, we observed that a reduced maximum FRI remainder degree did not impact the security level.

The final benchmark results, which collected the metrics for the STARK protocol when configured using a combination of the best-performing parameters, produced some disappointing results. The outcomes of this benchmark can be seen in Table 6.4. Each metric, except for the conjured security level, showed an improvement over the default configuration. While this is true, a closer examination reveals that the achieved metrics were worse than those achieved by just changing the FRI remainder maximum degree to 255. Only the proven security level improved when using this 'optimal' configuration as opposed to choosing the default configuration and altering the FRI remainder maximum degree to 255. We further reflect on this finding in chapter 7.

Option	**PS (B)**	**PT (ms)**	**VT (ms)**	**SC (b)**	**SP (b)**
NQ: 1	2015	1.864	0.019	2	15
NQ: 24	15985	2.581	0.118	71	49
NQ: 41	25137	1.912	0.195	116	73
NQ (D): 42	24866	11.339	0.199	116	74
NQ: 43	25361	9.592	0.193	116	75
NQ: 84	40497	2.722	0.351	116	87
NQ: 168	61759	4.103	0.573	116	87
NQ: 255	80226	2.355	0.820	116	87
BF: 2	16978	4.697	0.151	41	34
BF: 4	20952	1.250	0.177	99	55
BF (D): 8	24866	11.339	0.199	116	74
BF: 16	28532	3.804	0.211	115	84
BF: 32	33065	16.650	0.231	114	80
BF: 128	40778	29.176	0.254	112	73
GF: 0	24963	1.853	0.195	116	60
GF: 4	25507	1.874	0.200	116	64
GF: 8	23615	1.895	0.192	116	67
GF (D): 16	24866	11.339	0.199	116	74
GF: 20	25283	184.940	0.202	116	77
GF: 24	24513	2671.200	0.190	116	80
GF: 32	-	-	-	-	-
FFF: 2	33641	5.715	0.211	116	74
FFF: 4	28032	5.004	0.186	116	74
FFF (D): 8	24866	11.339	0.199	116	74
FFF: 16	28640	11.503	0.391	116	74
FRMD: 3	26628	5.325	0.235	116	74
FRMD: 7	26940	5.616	0.230	116	74
FRMD: 15	27835	6.441	0.247	116	74
FRMD (D): 31	24866	11.339	0.199	116	74
FRMD: 63	24014	5.051	0.194	116	74
FRMD: 127	25060	8.762	0.191	116	74
FRMD: 255	20099	2.420	0.165	116	74
Hash: Blake3_192	21328	6.327	0.201	96	74
Hash (D): Blake3_256	24866	11.339	0.199	116	74
Hash: SHA3_256	25235	41.229	0.400	116	74
FE (D): None	24866	11.339	0.199	116	74
FE: Quadratic	32196	5.966	0.354	128	76
FE: Cubic	-	-	-	-	-

Table 6.3: Results for option parameter changes in the STARK benchmark

PS (B)	**PT (ms)**	**VT (ms)**	**SC (b)**	**SP (b)**
23685	3.4192	0.17619	115	81

Table 6.4: zk-STARK combined configuration values benchmark

Chapter 7

Discussion

In this chapter, we discuss the research and benchmark performed as described in previous chapters. Starting in section 7.1, we discuss the results achieved from the benchmark, including a discussion on our findings as well as a general discussion on the implementation and the used ZKP protocol libraries. With the achieved results discussed, we aim to answer our research questions from section 1.2 in section 7.2. We continue the discussion by talking about the strengths of our research in section 7.3, and subsequently contrast these strengths by examining the limitations of our work in section 7.4. Finally, in section 7.5, we discuss the significance of our work and the potential use cases for the contained knowledge.

7.1 Achieved results

In our work, we benchmarked four general purpose NIZKP libraries implementing the zk-SNARK, zk-STARK, and Bulletproof protocols for in real applications. We benchmarked these libraries in an equal an equivalent application related to the privacy-preserving authentication context. From the benchmark results, detailed in chapter 6, we observed the following ordering between the protocols regarding proof size, proof generation time, and proof verification time:

- Proof size: We found that the SNARK protocol produced the smallest proofs, with the zk-STARK protocol producing the largest proofs. The Bulletproof implementation produced proofs that were somewhere in the middle, yet closer to the proof size from the SNARK. The Bulletproof proof size was within one order of magnitude from the two SNARK implementations, while the STARK implementation proof was at least one order of magnitude larger than the two other protocols. We note that this observation considers just the proof size, not including the verifying key size in the SNARK protocol.
- Proof generation time: Though with some fluctuations in the duration metrics, we overall observed the STARK implementation to be the fastest in generating a proof. The two SNARK implementations came in at the second place, with the proof times for these three implementations remaining within one order of magnitude difference. Generating a proof using the Bulletproof implementation took longer than for the other protocols, with a proof time that was more than an order of magnitude larger for the upper MiMC round numbers.

- Proof verification time: When verifying a proof, the STARK protocol performed the verification fastest. The Bulletproof proof verified the slowest, except at the lowest number of MiMC rounds where the proof verified slightly faster than the two SNARK proofs. Interestingly, the verification times for the STARK and Bulletproof proofs increased much more rapidly with the number of MiMC rounds than the SNARK proofs. While the STARK implementation was well over an order of magnitude faster at lower MiMC round numbers, this difference had shrunk to just around or even within an order of magnitude difference compared to the Go or Rust SNARK implementations, respectively, at the largest number of MiMC rounds. In the same way, the Bulletproof proof went from verifying slightly faster than the SNARK proofs at the lowest number of MiMC rounds, to verifying more than two orders of magnitude slower than the SNARK proof by the largest number of benchmarked MiMC rounds.

We relied on some other works for numbers to cross reference our results to. We found a few comparisons equating the three protocols, which all seemed to use the same metrics widely circulated around the internet. They were for example included in a presentation from ZCash [120], a Beanstalk network presentation [90], and even used as an image on the non-interactive zero-knowledge proof Wikipedia page [93]. We note that we could not, however, verify how these metrics were obtained nor on which application they were benchmarked. The slides from the two presentations did not provide any source where these values came from, and the Wikipedia page was missing a citation for the image. We included these metrics for reference in Table 7.1. Assuming the found metrics are valid, and disregarding that the hardware used to perform the benchmark is unknown, we cross referenced the metrics to our results obtained from the benchmark to observe that our results indicated a corresponding performance ordering for most metrics. The ordering for the proof size matched, and even the exact figures were comparable to the ones we obtained at higher numbers of MiMC rounds. We remark that it is not exactly meaningful that the exact metrics match, though, since we expect the found comparison to be obtained from an entirely different application benchmarked on different hardware. We therefore expect this correspondence to be coincidental. For the proof time, the ordering of the best performing protocols also matched, even with the SNARK and STARK metrics being much closer to each other than to the Bulletproof at higher MiMC round numbers. Only for the verification time, the ordering in our benchmark was different to the cross-reference source. Whereas in our benchmark the STARK implementation verified faster than the SNARK implementations, the cross-referenced comparison stated the inverse. What did match, however, was that the SNARK and STARK times were much closer together, with the Bulletproof proof verifying significantly slower. At least, when considering the results we obtained for larger numbers of MiMC rounds.

Protocol	**P (B)**	**PT (ms)**	**VT (ms)**
Bulletproof	1300	30000	1100
SNARK	288	2300	10
STARK	45000	1600	16

Table 7.1: Found external protocol comparison

Regarding the cross-check for the proof size, this only included the actual proof size. When we included the verification key as well, as required by the verifier to verify a proof in the two SNARK protocol implementations, the outcome changed. Not only did the Rust implementation in that case have a combined size almost as large as the proof size for the Bulletproof protocol, for lower MiMC round numbers, the total size of this data for the Go

SNARK implementation became larger than the Bulletproof proof. Not only that, but the combined size also furthermore became so large at higher numbers of MiMC rounds that the Go SNARK implementation had a larger combined verification key and proof size than the size of the STARK implementation proof. That was the case without even including the witness size, which the verifier additionally required in the Go SNARK implementation. Not only would including the verifying key in the comparison alter the performance ordering between the different protocols, but it also furthermore unveiled a clear contrast between the performance of two implementations of the same protocol. A contrast which manifested itself to a significantly smaller degree in the time-based metrics. We found this difference, a verifying key constant in size or almost increasing exponentially in size with the number of MiMC rounds, intriguing at the very least. While we aimed to limit such contrast between the different implementations of the three different protocols by using libraries written in the same programming language for each protocol, these observations not only tell us that that was the right thing to do, but also show the importance of optimized protocol libraries. Such optimization can make a substantial difference in the performance, even when both library implementations use the same Groth16 backend [63] underneath.

Lastly, we want to discuss the results achieved in the benchmark comparing the configuration parameter values for the zk-STARK protocol implementation. We examined the performance when configured using the settings that individually provided optimal performance, as described in section 5.4. We found that this improved the performance compared to our default configuration for all metrics except the conjured security level. We could argue that this means that we initially chose the wrong default configuration parameters. However, as mentioned in section 6.2, we achieved even better performance metrics when using the default configuration adjusting only the FRI maximum remainder degree. This demonstrated that the 'optimal' configuration parameter values when combined are not necessarily 'optimal' at all, and that the combination of different parameters forms a complex system of trade-offs. To truly inspect the impact of each parameter and the best performing configuration, in that case, would require more than benchmarking all combinations of parameters. Just to benchmark all combinations of our selected individual parameter changes would require benchmarking $8 * 6 * 7 * 4 * 7 * 3 * 3 = 84672$ configurations. Considering all parameter values would significantly increase this value. Even then, we would have benchmarked for just a single number of MiMC rounds, which as seen from our benchmark can significantly influence the performance of the STARK protocol implementation. And even at that point, we still would have only performed the benchmarks on a single hardware configuration, while different hardware configurations may benefit from different software configuration settings. Because of this, we still consider our approach of choosing the initial configuration using parameter values somewhere in the middle to be a safe choice, which enabled us to inspect the impact each parameter has on the protocol performance. In addition, we observed that the proof size, verification time, and conjured security level were not extremely different. Even the proof time, for which our default number of queries of 42 was a bad pick, reduced only six times by choosing 41 as the number of queries. While such performance improvement is not negligible, it is sufficiently within an order of magnitude difference even though it constitutes a larger improvement than the threefold improvement achieved by the combination benchmark. Given that the zk-STARK protocol had a proof size more than an order of magnitude larger than the second largest proof size created by the Bulletproof protocol, not to mention that the STARK implementation already showed the best performance for the proof time and proof verification time, a more optimal configuration would ultimately not have altered our conclusions. We therefore conclude that our findings are still valid, despite the sub-optimal default configuration that we used for the zk-STARK protocol.

7.2 Research question answers

Based on the achieved results, we can now attempt to answer the research questions from section 1.2. The two research questions stated for this work were:

1. What are the performance differences between the three included NIZKP protocols, as observed from a real-world implementation of each protocol in an application that is as equal as possible, expressed in efficiency and security level?
2. What use case contexts are most beneficial for each NIZKP protocol, given the unique combination of its features and performance metrics?

The first question we can conveniently answer for the performance by using Table 7.2, which includes the averaged performance for each protocol over the five benchmarks with different number of MiMC rounds. Important to note for this table is that we calculated the average using the original, exact, numbers, then rounded the average for the proof and verification times to three decimals. From this table we can clearly observe that the SNARK

Protocol	**P (B)**	**PT (ms)**	**VT (ms)**
Bulletproof	993.0	849.705	275.701
SNARK (Rust)	192.0	29.878	1.695
SNARK (Go)	484.0	18.155	2.351
STARK	28388.4	14.384	0.232

Table 7.2: Protocol comparison using the average performance over the five default benchmarks with different MiMC rounds

protocol generates the smallest proofs, whereas the generated proofs from the Bulletproof and STARK protocols are slightly larger or significantly larger, respectively. This proved to be a significant disparity with the proof and verification times, for which we observed the shortest average proof generation and verification times from the STARK protocol. The SNARK and Bulletproof protocols took longer to create and verify their proofs. This observation answers the research question regarding the performance aspect, yet it is not a comprehensive perspective on its own. The SNARK protocol, as implemented in our benchmark, required a trusted setup. There exist situations where this is not desirable, as it requires trust in the party that performs the setup. Similarly, the STARK protocol in our benchmark involved limitations using private data in the proofs, whereas for the Bulletproof protocol we did not apply some specific benefits not found in other protocols. We refer the reader to other sections in this chapter for more discussion on this aspect. Given the limited availability of security level metrics from the libraries we used to implement the benchmark applications, we were unfortunately, as likewise discussed in other sections in this chapter, unable to answer the security level component of this question. While other sources for these metrics indicated that the security level was comparative for the used configurations, this was no guarantee and would require additional research and implementation work to confirm.

The second research question we answer in detail through our recommendations in section 8.2. To summarize: The zk-SNARK protocol is a good overall choice for performance, granted that a trusted setup is conceivable for the specific use case. The small proof sizes make the protocol particularly beneficial for Internet of Things (IoT) usage, where notable storage, bandwidth, or processing power limitations apply. The Bulletproof protocol is a viable alternative for the zk-SNARK in these applications when a trusted setup is unacceptable and can furthermore be a great option for applications that require proofs that values lie within a pre-determined range. This suitability, however, comes at the cost of much larger proof creation and verification times, though the latter of can be reduced significantly when the application allows batching of proof verifications. The zk-STARK protocol, finally, is currently best applied to succinctly prove the correctness of computations. This makes the STARK protocol for example applicable to cloud computing and distributed learning applications. The STARK protocol allows to quickly generate a proof for large statements, and is even quicker in verifying the generated proofs, though there exists a significant trade-off in the substantial size of the generated proofs. Finally, the zk-STARK protocol is the only viable option when the quantum resistance of the protocols is an important requirement, given that the other two protocols use cryptographic primitives that are not quantum resistant.

With the research questions answered, we reflect on the aims and objectives from section 1.3 in which we presented the following research questions:

1. Create an implementation and evaluate the protocols in a practical setting, using a common benchmark for a real-world use case.
2. Create a comparison of the efficiency and security of these three protocols, including their trade-offs between efficiency and security.
3. Describe recommendations for the use of these protocols in different applications, based on their strengths and weaknesses.

Regarding the first objective, we proclaim that we fully achieved it considering that our benchmark indeed evaluated the protocols in a practical setting for a real-world use case. Regarding the second objective, while we were able to compare the efficiency of the three NIZKP protocols including their efficiency trade-offs, we were insufficiently able to do the same for the security aspects of the protocols. Given the limitations of the libraries that we used to benchmark the three protocols, we could only obtain the security level metrics from a single protocol. While this work did include an attempt to complement these metrics using expertise from works by other authors, this did not satisfy the comparison for the actual implementations that we had in mind. Somewhat consoling is our inclusion of the security primitives and limitations for each protocol in chapter 4, which provided alternative knowledge on the security of each protocol that should partly offset the limited security comparison in the practical setting. This aspect constitutes a potential direction for future research. The third objective, we adequately answer in section 8.2. While it was inconceivable to enumerate all potential applications best suited to each protocol, we believe that we provided a fair number of categories and applications that constitute a thriving environment for each protocol. We leave the ideation of other applications up to other researchers, which they can derive from the information conveyed in this work, with the potential for them to unearth entirely new, unprecedented, application categories.

7.3 Strengths

The main strength of this work lies in the benchmark procedure performed on the three main NIZKP protocols: zk-SNARK, zk-STARK, and Bulletproofs. The benchmark application that we implemented for this procedure was relevant to real-life applications focusing on privacy-preservation and authentication. Additionally, we performed the benchmark using four existing general purpose NIZKP libraries that allowed for general applicability in all kinds of zero-knowledge proof applications. This is an important aspect of our work, since these libraries enable using ZKPs in all kinds of applications without the extensive knowledge that would be required to securely realize a custom implementation for one of the NIZKP protocols. All together, this means that our benchmark provides a helpful indication of the performance differences between each ZKP protocol when utilized. To the best of our knowledge, our work constitutes the first research that directly compares the three main NIZKP protocols using results from an equivalent benchmark implemented with existing general purpose ZKP programming libraries. We argue that our decision to use general purpose NIZKP libraries increases the relevance of the obtained benchmark results for researchers aiming to implement an application, since the libraries allow researchers to implement a ZKP into their application faster and more securely without deep knowledge on the cryptography behind each protocol. In situations where the overhead of general purpose NIZKP libraries is known to be unacceptable, the exact ZKP protocol that one should use is undoubtedly known. In the unlikely event where this statement does not apply, the relative speed by which the general purpose NIZKP libraries allow to implement a ZKP will quickly surface this requirement from the proof-of-concept implementation. Affected researchers can then pivot to a custom NIZKP implementation, or different protocol altogether, without having wasted too much research time.

While in section 7.1 we detailed some metrics that float around on the internet comparing the three main NIZKP protocols, we were unable to find the source of these metrics. As a result, we could not determine which application they benchmarked and which hardware and software they used in the process. This left us with uncertainty regarding how the metrics were obtained. In contrast, one of the main strengths of our work is the detailed documentation of the benchmarking procedure. Not only does this enable other researchers reproduce our efforts, it furthermore allows them to extend this research work to fill additional knowledge gaps and advance knowledge on the topic of ZKPs.

Another strength of our work is that it not only provides a comparison benchmark between the three main NIZKP protocols, but it also describes the cryptographic primitives forming each protocol in chapter 4. This not only allow researchers to gain insights for the right ZKP protocol to use in their application regarding performance, but also provides them with a source for knowledge on the cryptographic primitives behind each of the ZKP protocols. From our perspective, this makes our work an ideal starting point for any researcher to obtain more knowledge on of the three NIZKP protocols, especially when they have the intent to utilize one of the three discussed NIZKP protocols for a privacy-preserving application.

7.4 Limitations

In view of the strengths as discussed in section 7.3, it is just as important to discuss the many limitations of this work. Discussing these limitations accentuates where our work leaves something to be desired, and where other researchers can step in to fill the knowledge gaps. Most of the limitations described in this section were a direct result of the scoping of the work and the decisions we made in the process. Some of these decisions were a compromise, where we deliberately chose to accept a limitation mentioned in this section to further increase one of the strengths of this work as mentioned in section 7.3.

The main limitation to this work is that the results obtained from the benchmark do not necessarily indicate the performance of only the protocol. The metrics partially reflect the performance of the ZKP implementation library, which may or may not be well optimized, and to a lesser degree that of the programming language in which it is written. This is a direct trade-off from our aim to benchmark a real-world implementation of an application using zero-knowledge proofs, which necessarily involved an implementation of each NIZKP protocol that can impact the performance. We further increased the impact of the implementation on the protocol performance through our decision to benchmark general purpose NIZKP libraries. While we justified this decision by stating that this is how most applications will implement ZKPs, through a general purpose NIZKP library that removes the extensive knowledge requirement for a custom implementation, it did mean that the obtained performance metrics were even further removed from the theoretical performance that the protocol could provide. We observed this impact first hand when inspecting and discussing the performance differences between the Rust and Go implementations of the zk-SNARK protocol. These two libraries showed vastly different performance, even while we ensured both used BLS12-381 elliptic curve [25] and the Groth16 backend [63]. To reduce the impact of this limitation, we decided early on to implement the benchmark using a library for each ZKP protocol written in the same programming language. As discussed in chapter 5, we chose the Rust language for this, while we also included a single library in another language as a means for comparison. The comparison enabled us to show, with numbers, how the library can impact the performance of a protocol, as discussed in section 7.1. While we expect this decision to have benefited the conclusiveness of the obtained benchmark results, we also admit that we cannot guarantee this. There are simply not enough libraries that implement zero-knowledge proof protocols to include multiple libraries written in the same programming language for the same ZKP protocol in this research. This is another limitation of our work, which other researchers have the potential to rectified in the future when alternative NIZKP libraries have emerged for each protocol. The comparison with metrics for each protocol circulating on the internet which we used to show that our benchmark achieved comparable results, however, contributed to our confidence that the overall performance observations from our benchmark were accurate despite these limitations.

Another limitation to our research resided in the implementation specifically for the Bulletproof protocol. This protocol allows for batch verification, in which the verifier verifies many proofs in a single batch. Such batch verification, through some optimizations made possible by the mathematics in the underlying cryptographic protocol, enables any additional proof in the same batch to be verified faster than the first one. In the Bulletproof paper [33], the authors gave some examples in which the batch verification of many proofs amortizes the verification time to a fraction of the cost required for verifying a single proof by itself. Furthermore, batch verification works even between proofs for different circuits, if the proofs used the same public parameters [33]. Even though the Bulletproof library crate that we used did implement batch verification [126], we designed our benchmark for creating and verifying a single proof. As such, we did not make use of the option for batch verification. This means that, given an application where batch verification is possible, the Bulletproof protocol may not be the slowest protocol in verification as was the case in our benchmark. The Groth16 backend from the SNARK protocol implementations allows for a similar batch verification optimization, which at least the Rust [9] library seems to implement currently. Like the Bulletproof protocol, however, we did not benchmark this. This means that also for the Rust zk-SNARK protocol the performance for verification may be better than stated, provided that the application can perform the verifications in batches. Furthermore, the Bulletproof and zk-SNARK protocols provide an option for proof aggregation. Aggregation combines multiple proofs into a single proof, which proves whether all included proofs in the aggregated proof are valid. While the aggregated proof is smaller than the size of each individual proof in total, aggregation of proofs does not come for free. It requires a multi-party computation protocol and additional aggregation time to combine the proofs. For the Bulletproof protocol, the original paper [33] included the possibility for aggregation, whereas Gailly et al. described the option for the zk-SNARK protocol to aggregate Groth16 proofs in their SnarkPack paper [55]. While aggregation is possible in theory, it seems not widely implemented in practice. Our selected SNARK libraries do not implement aggregation at all, while for the Bulletproof crate the implemented aggregation seems to only concerns range proofs [31]. Since we use R1CS Bulletproof proofs in our benchmark, we could not use said aggregation even if we wanted to. In short, not considering batch verification or aggregation is a limitation to our research. However, we argue that the requirement to implement aggregation in actual applications reduce the impact of this limitation, in addition to the restricted availability in NIZKP libraries in the first place.

Yet another limitation to this research work is the implemented application. Because we were unable to implement our initial benchmark application idea, for reasons described in chapter 3 section 3.2, we implemented a much more basic application. Not only is this more basic implementation only slightly related to a privacy-preserving authentication use case, but the implementation also has the possibility to leak information because of current limitations in the zk-STARK protocol library. This limitation is a direct consequence of our aim to keep the application for each protocol as equal as possible, and specifically means that our benchmark application implementation should not be used as a how-to guide on implementing ZKPs in production applications. Furthermore, our implementations for each protocol included the round constants as input. While hard coding these constants in the circuit could result in different performance metrics for some protocols, we did not do this to keep the implementation as similar as possible for each protocol. While this is another limitation to our work, we justify the limitations above by arguing that we required the corresponding choices to attain our research objectives.

This nicely prompts the next limitation, in which we argue that designing a single equivalent benchmark application for all three protocols is a limitation in itself. The basis for this argument lies in each protocol having a specific speciality in which it excels. For example, the speciality of the Bulletproof protocol is its range proofs. These range proofs allow it to prove that a value lies within a certain range more efficiently than possible in other protocols. As the Bulletproof paper notes, unlike other protocols, the Bulletproof protocol eliminated the need to implement a commitment opening algorithm in the verification circuit of range proofs [33]. However, we do not utilize this strength in our benchmark. Likewise, the Winterfell library implementing the zk-STARK protocol specifically notes that the focus of the library is on proof of computation, which is slightly different from the proof of knowledge from the SNARK and Bulletproof R1CS proofs. Specifically, unlike the Bulletproof, Bellman, and Gnark libraries, the Winterfell library does not differentiate between private and public inputs. From the project's readme file: "The current implementation provides succinct proofs but NOT perfect zero-knowledge. This means that, in its current form, the library may not be suitable for use cases where proofs must not leak any info about secret inputs.". This means that the Winterfell may be more useful to succinctly prove computations than to create proofs for improved privacy like the other two protocols. By forcing each protocol to implement the same application, we had to restrict our application to the lowest common denominator. To be specific, the requirement for an equivalent application implementation makes it so that we cannot use protocol specific traits. This not only meant that our benchmark did not benefit from special functionality in each protocol, like the batch verification described before, it also meant that we did not take advantage of potential performance improvements. All together, this is a limitation that we could not circumvent. Had we implemented different applications or specific alterations for each protocol implementation, then our benchmark may indeed have become more representative for each individual protocol. Yet at the same time, this would have eliminated our possibility to perform a fair comparison between the protocols. For our work, we deliberately decided that a fair comparison between the protocols was more important.

The final limitation is our ability to compare the security level of each protocol. As it turned out, only one protocol implementation provided us with functions to obtain the security level of the proof, something that we failed to consider when selecting the protocol libraries to use for our benchmark implementations. While we aimed to resolve this by providing some sort of comparison using references to theoretical security levels of the two SNARK protocol implementations, we did not find a similar work for the Bulletproof protocol. Not only did this cause an incomplete comparison on the security level of the proof from each protocol, but the comparison that we provided additionally compared a practical number with a theoretical one. We should note, though, that the security level metric retrieved from the zk-STARK protocol implementation is in some ways a theoretical number as well, even though it was calculated from numbers in the actual implementation of the protocol. This is not something we could influence given our chosen NIZKP libraries, nevertheless it was a big limitation to our protocol comparison. We could have reduced the impact of this limitation by specifically choosing the benchmarked NIZKP libraries to include such functions. This assumes, however, the feasibility for us to make this choice in the first place, considering the other established requirements.

In conclusion, while our research included a substantial number of limitations, these limitations were a result of us designing the benchmark to the best of our ability given the set goals and requirements. We believe that our research accomplished the set goals despite the listed limitations. Consequently, we consider the strengths as detailed in section 7.3 to outweigh the limitations specified in this section. By revealing and cataloguing the limitations to this work, we hope to enhance its usefulness for researchers, and to incite research into works that eliminate some of the limitations by taking other approaches and making different trade-offs.

7.5 Significance & potential applications

This work, to the best of our knowledge, is the first verifiable research work to compare the three NIZKP protocols zk-SNARK, zk-STARK, and Bulletproofs, in real-world application. While in section 5.2 of our previous SLR [95], we identified some works that also compared these protocols, they were different for a variety of reasons. As stated more thoroughly in that work, the two works most closely related to ours [60] [107] compared the three NIZKP protocols, yet remained in the theoretical realm and did not compare the real-world performance of these protocols. Another work by Panait et al. [96] focused on implementing libraries for the zk-SNARK and zk-STARK protocols yet did not include the Bulletproof protocol nor discussed the real-world implementation performances. While yet another work by Partala et al. [98] did assess the practical performance of included NIZKP protocols, compared to our work they did not include the zk-SNARK protocol, focused on applications in smart contracts, and collected only the asymptotic performance from surveyed works. This substantially differs from our work, which not only chose a more general privacy-preserving authentication related application, but furthermore benchmarked the direct performance metrics of the protocols from their implementation in an equivalent application. To summarize, we did not find any scientific research works that performed an identical benchmark comparing the three main NIZKP protocols. That is not to state that there were no comparisons between these three protocols out on the internet, however. As discussed in section 7.1, these comparisons do in fact exist [120] [90] [93]. Yet, it remains unclear how these metrics were collected. In this regard our work contributes reproducibility, improved transparency, and improved usability, by providing clarity on the context in which we obtained the results. Moreover, our benchmark applied general purpose protocol libraries to a widespread application, whereas we expect that the benchmark metrics floating around the Internet were extracted from custom protocol implementations in some cryptocurrency or blockchain related application. We argue that our work thereby ensures better applicability of findings in practice, since it is more practical to use general purpose libraries than to design custom protocol libraries for each specific purpose.

Considering all facets described in this section, we consider our work to constitute a novel contribution that benefits the corpus of scientific knowledge on the topic of zero-knowledge proofs.

We expect the knowledge and findings in this work to inspire many new applications and research works for zero-knowledge proofs, and to benefit both researchers and application designers by providing them with the required knowledge to decide on the correct NIZKP to utilize given the context of their use case. Next to that, we expect the knowledge from this work to benefit many communities. To specifically name one community, it should assist researchers in obtaining knowledge on NIZKP protocols and encourage research into the application of these protocols to many more types of applications, where they can provide privacy and security. To name some potential applications, we remark medical applications where user privacy is a high priority that currently limits applicability of some ideas, and trust-based applications including, for example digital voting or distributed computing, where the results must remain hidden but must also be provably valid to enable trust. We anticipate other researchers to be far more creative than us in thinking of many other uses, which will eventually benefit society at large. Ultimately, the usefulness of knowledge in this work is not limited to academic circles. It could also benefit commercial institutions in accelerating the development and acceptation of NIZKPs in general, again to the benefit of their users in society.

Chapter 8

Conclusion

In this chapter, we conclude our research in which we performed a benchmark for the zk-SNARK, zk-STARK, and Bulletproof ZKP protocols. First off, in section 8.1 we recollect the results from chapter 6 and reiterate our key findings. Following our key findings, we provide some recommendations on the utilization of NIZKPs that followed from our benchmark in section 8.2. Subsequently, we provide some promising future research directions on all kinds of NIZKP aspects that we would like to see realized in section section 8.3. In drawing things to a close, we finalize our work by providing a conclusion with some final remarks in section 8.4.

8.1 Key findings

In this chapter, we concisely reiterate the key takeaways from our NIZKP protocol benchmark. For more in depth findings, we refer the reader to chapter 6 and chapter 7, corresponding to the results and discussion chapters. We first recollect the results of the performance metrics found for all three NIZKP protocols, averaged over the five benchmarks on different numbers of hash rounds, listed in Table 7.2. From this table we clearly observed that the SNARK protocol generated the smallest proofs, while the STARK protocol generated by far the largest proofs. Regarding the proof generation and verification times, the STARK protocol was faster in both metrics than the two SNARK protocol implementations, while the Bulletproof protocol turned out to be by far the slowest for these metrics. We furthermore observed these findings to be analogous to the externally found protocol comparison for which we could not determine how they were benchmarked, included for reference in Table 7.1. The exception to this equivalence was the protocol ordering in the proof verification times between the SNARK and STARK, which switched place in our results. Given that the absolute difference between these reversed metrics was small for both our results and the external results, especially compared to the difference with the Bulletproof protocol, this does not constitute an alarming difference.

With all configuration settings in the zk-STARK protocol library, we found it sensible to benchmark the performance differences between these configurations. While we discovered that our default configuration may not have been optimal, we remarked that this realistically did not impact the conclusion from the comparisons between the protocols. Furthermore, we observed that the configuration parameter values which were individually optimal did not exactly provide the best possible performance when combined. We claimed this to be a result of the complexity of the inner working of the protocol and stated our suggestion to evaluate several configurations that fit the context when utilizing zk-STARKs in an application use case.

Regarding the security level of the protocols, we identified evidence that the performance on this aspect between the protocols did not deviate for our chosen configurations. With that said, this finding was inconclusive given that three of the four protocol implementing libraries did not include a method to obtain such security level metric. As such, we had to supplement our findings with complementary data from research works by other authors.

8.2 Recommendations

Reflecting on the obtained results from chapter 6, and the discussion that subsequently ensued in chapter 7, in this section we strive to provide some recommendations on which application contexts we would recommend utilizing each protocol.

We start with the zk-SNARK protocol. The two implementations for this protocol showed the smallest proof size, in addition to the proof size itself being constant. The small proof size makes this protocol a great contender for applications where either storage space is limited, or where the network connection has a restricted capacity or transfer speed. An example of a situation where storage space is limited is in blockchain systems, for which we can see zk-SNARK protocol already in use in e.g. ZCash [6]. Limited network connections, on the other hand, are a reality for Low Power Wide Area Networks (LPWANs), often used in Internet of Things (IoT) applications and sensor networks where the devices are in a remote location and have low power requirements [110]. The small and constant size of the SNARK proofs, especially those created by the Rust implementation, make the zk-SNARK protocol a good protocol to consider for these kinds of applications. Furthermore, as benchmarked, creating a SNARK proof is not much more compute intensive than creating a STARK proof, which is beneficial for the IoT application where devices and sensors are often low powered devices with little compute power. The most important consideration to make before applying the zk-SNARK protocol, even for these applications, is whether the requirement for a trusted setup is acceptable. There are sparks of hope to apply the zk-SNARK protocol in situations where a trusted setup is unacceptable. Researchers have recently created new SNARK backend techniques, including SuperSonic [32] and Halo [27], that do not require a trusted setup in certain situations. ZCash currently uses a Halo 2 zk-SNARK backend [26] in their network, which according to them eliminates the trusted setup requirement. As it currently stands, however, the trusted setup is a definite requirement in the Groth16 backend implementation used by both the Rust and Go zk-SNARK protocol libraries benchmarked in this work. We therefore recommend investigating the use of the zk-SNARK protocol for applications where the proof size is a key factor, including blockchain and IoT applications, yet to ensure that the trusted setup requirement to obtain a CRS is not a hindrance in said application.

For applications in which a trusted setup is not an option, the Bulletproof protocol offers a viable alternative. Bulletproof proofs are not considerably larger than SNARK proofs, especially when compared to STARK proofs. Unlike the SNARK proofs, though, the size of the Bulletproof proofs is not constant. A further downside for the applicability of the Bulletproof protocol are the much larger proof creation and verification times than in the two other protocols, which furthermore increase more rapidly as well with the size of the computation. At present, this makes the Bulletproof protocol less suitable to apply to low compute IoT environments. In applications where aggregation of proof and batch verification, as discussed in chapter 7 section 7.4, is possible, the proof size and especially the verification times can however be significantly reduced. This is beneficial in situations where a single prover must create a proof, but many verifiers need to verify that proof. This applies for example when proving and verifying transactions in blockchains, for which e.g. the Monero network [88] already applies the Bulletproofs protocol. The Bulletproof protocol has yet another benefit, not visible in our benchmark since we use R1CS proofs, in that is specializes in range proofs. This allows the Bulletproof protocol to be especially beneficial and performant in applications which use ZKPs to prove that a certain value lies within a pre-determined interval. In general, applications that benefit from such range proofs include financial transactions, income checks, and age verification. There are, however, many more specialized uses for range proofs, including genomic range queries [69]. In brief, our recommendation is that the Bulletproof protocol could be a viable alternative to the SNARK protocol in situations where a trusted setup is undesirable, and where the proof creation cost is not a limiting factor, or where a proof is verified frequently after it is created once. We furthermore recommend investigating the use of the Bulletproof protocol specifically where the proof must prove that a value is inside of a pre-determined range, a use case which Bulletproof range proofs are particularly good at.

Finally, there is the zk-STARK protocol. Given the proof size which, in our benchmark, was at least an order or magnitude larger than that for the other two protocols, we can only recommend the use of the STARK protocol for applications where the proof size is not important. An example where the proof size is unlikely to be important is in the context of cloud computing, datacentres, or machine learning. In that application context, ample storage space and network capacity is available, and datasets used as input to calculations can be extremely large to begin with. In return for the large proof sizes, we observed a low proof creation time and especially short proof verification time compared to the other protocols in our benchmark. These small proof and verification times become especially useful when applied to large computations as performed in datacentres and machine learning. This applicability factors into the zk-STARK protocol in general, and to an even greater degree for the Winterfell library used in our benchmark. Currently, this library does not implement perfect zero-knowledge, instead the library aims to enable succinctly proving computations instead. This makes it hard to securely implement applications where the proof proves a statement on confidential data, as the generated proof could leak this data. This is a significant distinction from the Bulletproof and zk-SNARK protocol implementations, which do intend to guard against the verifier obtaining confidential information. For reasons listed above, we recommend considering the zk-STARK protocol, and specifically the Winterfell library, in situations where the application uses ZKPs to ensure the correct execution of a computation in a succinct manner. This includes, but is not limited to, machine learning, distributed or multi-party computations, and verifiable computing applications, e.g. in the cloud.

This brings us to our final advise when contemplating which NIZKP protocol and library to use for a given application context. We recommend to, where possible, create a proof of concept for the desired application using multiple libraries implementing the same protocol. When in doubt between multiple protocols, try them all in a way that is representative yet does not cost a lot of time. This recommendation stems from two observations: first, the challenges we had in applying the three protocols to a single, equivalent, application. Second, the Rust and Go libraries both implementing same Groth16 SNARK protocol [63] yet exhibiting different performance metrics, particularly regarding the size of the proving and verifying keys in the CRS. We furthermore not only recommend trying out multiple protocols and multiple libraries for the same protocol, but we also advocate to attempt different methods to utilize ZKPs in the application. Specifically, when using the STARK protocol, we furthermore recommend evaluating the performance for several configurations to see which best achieves a pre-determined set of objectives for the application.

All these tests can lead to vastly different performance metrics, which could make or break the usability of NIZKPs in an application context. While we understand that this recommendation requires a considerable time investment, we hope that our work can reduce this time investment by serving as knowledge base to limit the amount of experimentation required to find the right NIZKP protocol that best fits the application needs.

8.3 Future directions

With the results, discussion, strengths, limitations, and recommendations out of the way, we will now provide some suggestions for future research directions.

First, we would like to suggest research which compares many different programming libraries implementing the same NIZKP protocol. These libraries could be written in different programming languages, as long as the implemented protocol is the same. This would not only better indicate the differences between several libraries than we did in our comparison, since that was not our main goal, it would also provide a nice overview for anyone wanting to implement a given protocol in an application using a library. The comparison could not only compare the performance of the protocols, but also the features that each implementation includes. In addition, a comparison of different libraries implementing an identical protocol would have an easier time implementing a more detailed and interesting application for the benchmark. The direct result of such benchmark would be that it provides visibility to the specialization of the protocol more than our benchmark did. We believe that research performing the described comparison is valuable to read for anyone that has the goal to utilize that specific NIZKP protocol in any given application.

Second, we think it would be interesting for future research to examine whether our initial benchmark application idea of implementing zkAttest, as introduced by Faz-Hernández et al. [50], for all three NIZKP protocols would be doable after all. Our research as described did not have the capacity to implement this application, yet any research could easily extend our current benchmark with the results of a benchmark for such application. Such addition would provide an even better idea of the real-world performance to expect from each protocol and matching libraries.

Third, we believe there is room for more research into new and improved NIZKP protocols. Researchers have performed vast amounts of research on NIZKP protocols in the past few years, with the Bulletproof protocol [33] and FRI underlying the STARK protocol [12] originating only in 2017, while work on the zk-SNARK protocol has not been dormant either with the introduction of the Sonic [81], SuperSonic [32], Halo [27], and Halo 2 [26]. Even the Groth16 SNARK scheme [63], which originated in 2016 and is widely implemented in SNARK libraries, is continuously improved upon with for example the in chapter 7 section 7.4 mentioned work by Gailly et al. [55] from 2021 which introduced aggregation for Groth16 proofs. As we found in this research, however, in practice implementations understandably lag research. Furthermore, there is still a vast number of limitations and performance implications that anyone utilizing NIZKPs to prove knowledge or computations in their application must deal with. We expect that future research works can resolve more of these limitations, which would open opportunities to gain benefits from using the ZKP protocols in applications without the current downsides. For this reason, we argue that more research on NIZKP protocol improvements would benefit for the ZKP ecosystem.

Fourth, as mentioned in the limitations to our work in chapter 7, our work was unable to compare in detail the actual security level of most of the benchmarked protocol implementations. This leaves us with questions on which of the three protocols is most secure.

Therefore, we indicate this aspect could be researched in-depth in a future work.

Fifth and last, we recommend a future research direction into the establishment of benchmarking standards for ZKP applications. We anticipate that introducing such standard would make it easier to compare research on applications implementing ZKPs, when the authors of these works benchmarked their application and followed the set standard while doing so. We furthermore anticipate that an established benchmark standard would entice implementing libraries to implement functionality to obtain the metrics defined in this benchmarking standard, which would make it even easier for researchers that implement an application using such library to include the standardized ZKP metrics for comparison. While we do not expect a standard to be all-encompassing, nor do we expect every researcher to embrace it, we would still consider it an improvement over the current situation in which comparing the performance of ZKP protocols in applications is a complex endeavour.

8.4 Conclusion

In this research, we designed and implemented a benchmark to compare the three NIZKP protocols, zk-SNARK, zk-STARK and Bulletproofs, in a real-world setting. To achieve this, we designed a single benchmark application that incorporates privacy-preserving authentication uses. The application we decided on, after deliberating some other options, was to implement a MiMC hash with a variable number of rounds. After describing the methodology for this work, we provided a concise description of the mathematical primitives underlying each protocol. This description included the security assumptions they made, as well as the vulnerabilities and limitations present in each. By providing this information we aimed to supply readers with sufficient information to understand the basic workings that enabled their functionality and established their characteristics. By additionally describing previous ZKP vulnerabilities and how to prevent or resolve them, we strengthened the idea that deciding which protocol to use is not always a performance related proposition. Our intention for this was to reinforce the notion that security and privacy are central in implementing NIZKP protocols in actual production ready applications. With the primitives clarified, we commenced by implementing the benchmark application. We implemented the application equally for each protocol using existing general purpose NIZKP libraries, namely Bellman [7] for the SNARK protocol, Winterfell [121] for the STARK protocol, and Bulletproofs [29] for the Bulletproof protocol. All three libraries were written in the Rust programming language. On top of that, we implemented the same application using the Gnark zk-SNARK library [58] written in the Go programming language. We decided on this additional implementation to compare the performance differences between two NIZKP libraries implementing the same protocol yet written in a different programming language. We benchmarked all implementations using a default configuration. Afterwards, we benchmarked just the zk-STARK protocol, altering a single configuration parameter at a time. Inspecting the results then allowed us to determine the performance impact of altering this parameter. The resulting from conducting the benchmark indicated the following performance characteristics: The SNARK protocol proofs were the smallest, in addition to being constant. The Bulletproof proofs were slightly larger, whereas the STARK protocol created by far the largest proofs. Neither the Bulletproof nor the STARK proofs were constant in size, and both increased with the number of hash rounds. The proof times for the SNARK and STARK protocols were comparable, with the STARK creating a proof faster overall. The Bulletproof protocol was much slower in creating proofs, which only worsened with an increasing number of hash rounds. We observed a similar pattern to the proof creation for the verification times, with the remark that we did not apply any form of batch verification in our benchmark. In the following chapters we discussed the collected results and described the strengths and limitations of our research. While our research had several limitations, we argued that these resulted from the choices we had to make for our benchmark, and that these limitations did not invalidate the results. Moreover, the strengths resulting from those decisions outweighed the induced limitations. In the last chapter of this work, we wrapped up our research by providing recommendations on the strengths of each benchmarked protocol and described the application contexts in which each protocol would prosper. We explained that the SNARK protocol would be the best protocol to in applications that benefit from small proofs, when the requirement for a trusted setup is not a critical issue. In situations where a trusted setup is undesirable, the Bulletproof protocol provides similarly sized proofs, at the cost of a higher proof creation and verification time. The Bulletproof protocol is furthermore beneficial for its specialization in range proofs, though we only benchmarked Bulletproof R1CS proofs in this work. Finally, we found the zk-STARK protocol to be most advantageous in application categories where large proof sizes are not a problem, whereas quick proof generation and verification times are convenient. We indicated that verifiable computation and machine learning are examples of such application categories, which the Winterfell library cemented by focusing on succinct proofs of computation, unlike the other two protocol libraries.

Ultimately, we expect our research to be useful for anyone looking into the use of non-interactive zero-knowledge proofs for some application. We consider our work to be an excellent starting point from which to obtain knowledge on the mathematical and cryptographic primitives that formed the three main NIZKP protocols and their analogous real-world performance aspects to consider.

Bibliography

[1] *112-bit prime ECDLP solved!* July 15, 2009. URL: https://web.archive.org/web/20090715060838/http://lacal.epfl.ch/page81774.html (visited on 01/25/2024).

[2] Martin Albrecht et al. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity.* Publication info: A minor revision of an IACR publication in ASIACRYPT 2016. 2016. URL: https://eprint.iacr.org/2016/492 (visited on 03/05/2024).

[3] *Anatomy of a STARK, Part 3: FRI.* Anatomy of a STARK. 2023. URL: https://aszepieniec.github.io/stark-anatomy/fri.html (visited on 01/09/2024).

[4] Diego F. Aranha, Youssef El Housni, and Aurore Guillevic. *A survey of elliptic curves for proof systems.* Publication info: Published elsewhere. Designs, codes and Cryptography. 2022. URL: https://eprint.iacr.org/2022/586 (visited on 03/08/2024).

[5] Aztec. *Disclosure of recent vulnerabilities.* Disclosure of recent vulnerabilities. Jan. 11, 2022. URL: https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities (visited on 11/07/2023).

[6] Aritra Banerjee, Michael Clear, and Hitesh Tewari. "Demystifying the Role of zk-SNARKs in Zcash". In: *2020 IEEE Conference on Application, Information and Network Security (AINS).* Nov. 17, 2020, pp. 12–19. DOI: 10.1109/AINS50155.2020.9315064. arXiv: 2008.00881[cs]. URL: http://arxiv.org/abs/2008.00881 (visited on 04/09/2024).

[7] *bellman - crates.io: Rust Package Registry.* Mar. 20, 2023. URL: https://crates.io/crates/bellman/0.14.0 (visited on 03/15/2024).

[8] *bellman-examples/src/sharkmimc.rs at master · lovesh/bellman-examples.* GitHub. Dec. 27, 2018. URL: https://github.com/lovesh/bellman-examples/blob/master/src/sharkmimc.rs (visited on 02/15/2024).

[9] *bellman::groth16::batch - Rust.* 2023. URL: https://docs.rs/bellman/0.14.0/bellman/groth16/batch/index.html (visited on 04/08/2024).

[10] Eli Ben-Sasson. *elibensasson/libSTARK.* original-date: 2018-02-28T08:50:22Z. Mar. 14, 2024. URL: https://github.com/elibensasson/libSTARK (visited on 03/25/2024).

[11] Eli Ben-Sasson, Lior Goldberg, and David Levit. *STARK Friendly Hash – Survey and Recommendation.* Publication info: Preprint. MINOR revision. 2020. URL: https://eprint.iacr.org/2020/948 (visited on 02/15/2024).

[12] Eli Ben-Sasson et al. *Fast Reed-Solomon Interactive Oracle Proofs of Proximity.* ISSN: 1433-8092. Sept. 8, 2017. URL: https://eccc.weizmann.ac.il/report/2017/134/ (visited on 12/07/2023).

[13] Eli Ben-Sasson et al. *Proximity Gaps for Reed-Solomon Codes*. Publication info: Published elsewhere. Minor revision. FOCS 2020. 2020. URL: https://eprint.iacr.org/2020/654 (visited on 12/07/2023).

[14] Eli Ben-Sasson et al. *Scalable, transparent, and post-quantum secure computational integrity*. Publication info: Preprint. MINOR revision. 2018. URL: https://eprint.iacr.org/2018/046 (visited on 10/31/2023).

[15] Eli Ben-Sasson et al. "Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. ISSN: 2375-1207. May 2015, pp. 287–304. DOI: 10.1109/SP.2015.25. URL: https://ieeexplore.ieee.org/abstract/document/7163032 (visited on 01/23/2024).

[16] Eli Ben-Sasson et al. "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge". In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 90–108. ISBN: 978-3-642-40084-1. DOI: 10.1007/978-3-642-40084-1_6.

[17] Eli Ben-Sasson et al. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Publication info: Published elsewhere. Minor revision. USENIX Security 2014. 2013. URL: https://eprint.iacr.org/2013/879 (visited on 01/23/2024).

[18] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. Jan. 22, 2017. URL: https://safecurves.cr.yp.to/ (visited on 01/24/2024).

[19] Nir Bitansky et al. *From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again*. Publication info: Published elsewhere. Unknown where it was published. 2011. URL: https://eprint.iacr.org/2011/443 (visited on 10/31/2023).

[20] Ian F. Blake, Maximilian Janisch, and Ulf Rehmann. *Berlekamp-Massey algorithm - Encyclopedia of Mathematics*. July 1, 2020. URL: https://encyclopediaofmath.org/index.php?title=Berlekamp-Massey_algorithm (visited on 12/19/2023).

[21] *bls12_381 - Rust*. Feb. 27, 2023. URL: https://docs.rs/bls12_381/latest/bls12_381/ (visited on 04/02/2024).

[22] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-interactive zero-knowledge and its applications". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. STOC '88. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1988, pp. 103–112. ISBN: 978-0-89791-264-8. DOI: 10.1145/62212.62222. URL: https://dl.acm.org/doi/10.1145/62212.62222 (visited on 01/22/2024).

[23] Dan Boneh et al. *Efficient polynomial commitment schemes for multiple points and polynomials*. Publication info: Preprint. MINOR revision. 2020. URL: https://eprint.iacr.org/2020/081 (visited on 01/11/2024).

[24] Jonathan Bootle et al. *Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting*. Publication info: A minor revision of an IACR publication in EUROCRYPT 2016. 2016. URL: https://eprint.iacr.org/2016/263 (visited on 10/31/2023).

[25] Sean Bowe. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. Electric Coin Company. Mar. 11, 2017. URL: https://electriccoin.co/blog/new-snark-curve/ (visited on 04/03/2024).

[26] Sean Bowe. *Explaining Halo 2*. Electric Coin Company. Sept. 1, 2020. URL: https://electriccoin.co/blog/explaining-halo-2/ (visited on 04/09/2024).

[27] Sean Bowe, Jack Grigg, and Daira Hopwood. *Recursive Proof Composition without a Trusted Setup*. Publication info: Preprint. MINOR revision. 2019. URL: https://eprint.iacr.org/2019/1021 (visited on 04/09/2024).

[28] *bulletproof-js*. npm. Jan. 31, 2020. URL: https://www.npmjs.com/package/bulletproof-js (visited on 03/25/2024).

[29] *bulletproofs - crates.io: Rust Package Registry*. Feb. 3, 2021. URL: https://crates.io/crates/bulletproofs/4.0.0 (visited on 03/15/2024).

[30] *bulletproofs-r1cs-gadgets/src/gadget_mimc.rs at master · lovesh/bulletproofs-r1cs-gadgets*. GitHub. May 27, 2019. URL: https://github.com/lovesh/bulletproofs-r1cs-gadgets/blob/master/src/gadget_mimc.rs (visited on 03/27/2024).

[31] *bulletproofs::generators::AggregatedGensIter - Rust*. Feb. 3, 2021. URL: https://doc-internal.dalek.rs/bulletproofs/generators/struct.AggregatedGensIter.html (visited on 04/08/2024).

[32] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. *Transparent SNARKs from DARK Compilers*. Publication info: A major revision of an IACR publication in EUROCRYPT 2020. 2019. URL: https://eprint.iacr.org/2019/1229 (visited on 04/09/2024).

[33] Benedikt Bünz et al. *Bulletproofs: Short Proofs for Confidential Transactions and More*. Publication info: Published elsewhere. Minor revision. 39th IEEE Symposium on Security and Privacy 2018. 2017. URL: https://eprint.iacr.org/2017/1066 (visited on 10/31/2023).

[34] Vitalik Buterin. *STARKs, Part I: Proofs with Polynomials*. Nov. 9, 2017. URL: https://vitalik.ca/general/2017/11/09/starks_part_1.html (visited on 10/31/2023).

[35] Lily Chen et al. *Recommendations for Discrete Logarithm-based Cryptography:: Elliptic Curve Domain Parameters*. NIST SP 800-186. Gaithersburg, MD: National Institute of Standards and Technology, Feb. 3, 2023, NIST SP 800–186. DOI: 10.6028/NIST.SP.800-186. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf (visited on 01/24/2024).

[36] R. Chien. "Cyclic decoding procedures for Bose- Chaudhuri-Hocquenghem codes". In: *IEEE Transactions on Information Theory* 10.4 (Oct. 1964). Conference Name: IEEE Transactions on Information Theory, pp. 357–363. ISSN: 1557-9654. DOI: 10.1109/TIT.1964.1053699. URL: https://ieeexplore.ieee.org/document/1053699 (visited on 12/19/2023).

[37] Information Technology Laboratory Computer Security Division. *Post-Quantum Cryptography | CSRC | CSRC*. CSRC | NIST. Jan. 3, 2017. URL: https://csrc.nist.gov/projects/post-quantum-cryptography (visited on 08/08/2023).

[38] *Consensys/gnark*. original-date: 2020-02-24T16:08:21Z. Mar. 15, 2024. URL: https://github.com/Consensys/gnark (visited on 03/15/2024).

[39] *Consensys/gnark-crypto*. Apr. 2, 2024. URL: https://github.com/Consensys/gnark-crypto (visited on 04/02/2024).

[40] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1965-0178586-1. URL: https://www.ams.org/mcom/1965-19-090/S0025-5718-1965-0178586-1/ (visited on 12/18/2023).

[41] *curve25519_dalek_ng - Rust*. Sept. 16, 2021. URL: https://docs.rs/curve25519-dalek-ng/latest/curve25519_dalek_ng/ (visited on 04/02/2024).

[42] Dana Dachman-Soled et al. *On the (In)security of the Fiat-Shamir Paradigm, Revisited*. Publication info: Published elsewhere. A merged version of this work and a work of [Bitansky, Garg, Wichs] will appear at TCC 2013. 2012. URL: https://eprint.iacr.org/2012/706 (visited on 01/22/2024).

[43] Gaby G. Dagher et al. *Provisions: Privacy-preserving proofs of solvency for Bitcoin exchanges*. Publication info: Published elsewhere. Major revision. ACM CCS. 2015. URL: https://eprint.iacr.org/2015/1008 (visited on 01/25/2024).

[44] Ivan Damgård. "Commitment Schemes and Zero-Knowledge Protocols". In: *Lectures on Data Security: Modern Cryptology in Theory and Practice*. Ed. by Ivan Bjerre Damgård. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 63–86. ISBN: 978-3-540-48969-6. DOI: 10.1007/3-540-48969-X_3. URL: https://doi.org/10.1007/3-540-48969-X_3 (visited on 12/20/2023).

[45] Xiaoyang Dong et al. "Quantum Attacks on Hash Constructions with Low Quantum Random Access Memory". In: *Advances in Cryptology – ASIACRYPT 2023*. Ed. by Jian Guo and Ron Steinfeld. Lecture Notes in Computer Science. Singapore: Springer Nature, 2023, pp. 3–33. ISBN: 978-981-9987-27-6. DOI: 10.1007/978-981-99-8727-6_1.

[46] William Easttom. "Cryptographic Hashes". In: *Modern Cryptography: Applied Mathematics for Encryption and Information Security*. Ed. by William Easttom. Cham: Springer International Publishing, 2021, pp. 205–224. ISBN: 978-3-030-63115-4. DOI: 10.1007/978-3-030-63115-4_9. URL: https://doi.org/10.1007/978-3-030-63115-4_9 (visited on 12/07/2023).

[47] *EdDSA | gnark*. Mar. 2, 2023. URL: https://docs.gnark.consensys.io/Tutorials/eddsa (visited on 03/27/2024).

[48] *ethSTARK documentation - Measuring Security*. Aug. 5, 2020. URL: https://github.com/starkware-libs/ethSTARK?tab=readme-ov-file#7-Measuring-Security.

[49] *facebook/winterfell*. original-date: 2021-04-23T19:20:43Z. Mar. 15, 2024. URL: https://github.com/facebook/winterfell (visited on 03/15/2024).

[50] Armando Faz-Hernández, Watson Ladd, and Deepak Maram. *ZKAttest: Ring and Group Signatures for Existing ECDSA Keys*. Publication info: Published elsewhere. Selected Areas in Cryptography – SAC 2021. 2021. URL: https://eprint.iacr.org/2021/1183 (visited on 03/22/2024).

[51] Amos Fiat and Adi Shamir. “How To Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology — CRYPTO’ 86*. Ed. by Andrew M. Odlyzko. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1987, pp. 186–194. ISBN: 978-3-540-47721-1. DOI: 10.1007/3-540-47721-7_12.

[52] G. Forney. “On decoding BCH codes”. In: *IEEE Transactions on Information Theory* 11.4 (Oct. 1965). Conference Name: IEEE Transactions on Information Theory, pp. 549–557. ISSN: 1557-9654. DOI: 10.1109/TIT.1965.1053825. URL: https://ieeexplore.ieee.org/document/1053825 (visited on 12/19/2023).

[53] Ariel Gabizon. *Explaining SNARKs Part I: Homomorphic Hidings*. Electric Coin Company. Feb. 28, 2017. URL: https://electriccoin.co/blog/snark-explain/ (visited on 11/02/2023).

[54] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Publication info: Preprint. 2019. URL: https://eprint.iacr.org/2019/953 (visited on 02/02/2024).

[55] Nicolas Gailly, Mary Maller, and Anca Nitulescu. *SnarkPack: Practical SNARK Aggregation*. Publication info: Preprint. MINOR revision. 2021. URL: https://eprint.iacr.org/2021/529 (visited on 04/08/2024).

[56] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. “Pairings for cryptographers”. In: *Discrete Applied Mathematics*. Applications of Algebra to Cryptography 156.16 (Sept. 6, 2008), pp. 3113–3121. ISSN: 0166-218X. DOI: 10.1016/j.dam.2007.12.010. URL: https://www.sciencedirect.com/science/article/pii/S0166218X08000449 (visited on 11/09/2023).

[57] Adam Gibson. *From Zero (Knowledge) to Bulletproofs*. https://github.com/AdamISZ/from0k2bp/blob
June 27, 2022. URL: https://github.com/AdamISZ/from0k2bp/blob/master/from0k2bp.pdf (visited on 10/31/2023).

[58] *gnark package - github.com/consensys/gnark - Go Packages*. Oct. 16, 2023. URL: https://pkg.go.dev/github.com/consensys/gnark@v0.9.1 (visited on 03/15/2024).

[59] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM Journal on Computing* 18.1 (Feb. 1989). Publisher: Society for Industrial and Applied Mathematics, pp. 186–208. ISSN: 0097-5397. DOI: 10.1137/0218012. URL: https://epubs.siam.org/doi/10.1137/0218012 (visited on 01/21/2024).

[60] Yinjie Gong et al. “Analysis and comparison of the main zero-knowledge proof scheme”. In: *2022 International Conference on Big Data, Information and Computer Network (BDICN)*. 2022 International Conference on Big Data, Information and Computer Network (BDICN). Jan. 2022, pp. 366–372. DOI: 10.1109/BDICN55575.2022.00074.

[61] Dan Gordon. “Discrete Logarithm Problem”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 352–353. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_445. URL: https://doi.org/10.1007/978-1-4419-5906-5_445 (visited on 11/28/2023).

[62] Lorenzo Grassi et al. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. Publication info: Published elsewhere. USENIX Security '21. 2019. URL: https://eprint.iacr.org/2019/458 (visited on 02/16/2024).

[63] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Publication info: A minor revision of an IACR publication in EUROCRYPT 2016. 2016. URL: https://eprint.iacr.org/2016/260 (visited on 11/10/2023).

[64] Jens Groth. "Short Pairing-Based Non-interactive Zero-Knowledge Arguments". In: *Advances in Cryptology - ASIACRYPT 2010*. Ed. by Masayuki Abe. Red. by David Hutchison et al. Vol. 6477. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 321–340. ISBN: 978-3-642-17372-1. DOI: 10.1007/978-3-642-17373-8_19. URL: http://link.springer.com/10.1007/978-3-642-17373-8_19 (visited on 10/31/2023).

[65] Lov K. Grover. *A fast quantum mechanical algorithm for database search*. Nov. 19, 1996. DOI: 10.48550/arXiv.quant-ph/9605043. arXiv: quant-ph/9605043. URL: http://arxiv.org/abs/quant-ph/9605043 (visited on 01/23/2024).

[66] Ulrich Haböck. *A summary on the FRI low degree test*. Publication info: Preprint. 2022. URL: https://eprint.iacr.org/2022/1216 (visited on 01/09/2024).

[67] R. W. Hamming. "Error detecting and error correcting codes". In: *The Bell System Technical Journal* 29.2 (Apr. 1950). Conference Name: The Bell System Technical Journal, pp. 147–160. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1950.tb00463.x. URL: https://ieeexplore.ieee.org/document/6772729 (visited on 12/19/2023).

[68] Darrel Hankerson and Alfred Menezes. "Elliptic Curve Discrete Logarithm Problem". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 397–400. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_246. URL: https://doi.org/10.1007/978-1-4419-5906-5_246 (visited on 11/28/2023).

[69] Seoyeon Hwang, Ercan Ozturk, and Gene Tsudik. "Balancing Security and Privacy in Genomic Range Queries". In: *ACM Transactions on Privacy and Security* 26.3 (Mar. 13, 2023), 23:1–23:28. ISSN: 2471-2566. DOI: 10.1145/3575796. URL: https://dl.acm.org/doi/10.1145/3575796 (visited on 03/31/2023).

[70] Yael Tauman Kalai, Guy N. Rothblum, and Ron D. Rothblum. *From Obfuscation to the Security of Fiat-Shamir for Proofs*. Publication info: Preprint. MINOR revision. 2016. URL: https://eprint.iacr.org/2016/303 (visited on 01/22/2024).

[71] Bhushan Kapoor and Pramod Pandya. "Chapter 2 - Data Encryption". In: *Cyber Security and IT Infrastructure Protection*. Ed. by John R. Vacca. Boston: Syngress, Jan. 1, 2014, pp. 29–73. ISBN: 978-0-12-416681-3. DOI: 10.1016/B978-0-12-416681-3.00002-1. URL: https://www.sciencedirect.com/science/article/pii/B9780124166813000021 (visited on 12/07/2023).

[72] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Advances in Cryptology - ASIACRYPT 2010*. Ed. by Masayuki Abe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 177–194. ISBN: 978-3-642-17373-8. DOI: 10.1007/978-3-642-17373-8_11.

[73] Askold Khovanskii, Sushil Singla, and Aaron Tronsgard. *Interpolation Polynomials and Linear Algebra*. Feb. 26, 2022. arXiv: 2203.01822[math]. URL: http://arxiv.org/abs/2203.01822 (visited on 01/30/2024).

[74] Dima Kogan. “Lecture 5: Proofs of Knowledge, Schnorr’s protocol, NIZK”. In: (2019). URL: https://crypto.stanford.edu/cs355/19sp/lec5.pdf.

[75] Watson Ladd. *Introducing Zero-Knowledge Proofs for Private Web Attestation with Cross/Multi-Vendor Hardware*. The Cloudflare Blog. Aug. 12, 2021. URL: https://blog.cloudflare.com/introducing-zero-knowledge-proofs-for-private-web-attestation-with-cross-multi-vendor-hardware (visited on 03/22/2024).

[76] *Lagrange interpolation formula - Encyclopedia of Mathematics*. June 5, 2020. URL: https://encyclopediaofmath.org/index.php?title=Lagrange_interpolation_formula (visited on 11/09/2023).

[77] Harashta Tatimma Larasati and Howon Kim. “Quantum Cryptanalysis Landscape of Shor’s Algorithm for Elliptic Curve Discrete Logarithm Problem”. In: *Information Security Applications*. Ed. by Hyoungshick Kim. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 91–104. ISBN: 978-3-030-89432-0. DOI: 10.1007/978-3-030-89432-0_8.

[78] Rolf Lindemann, Davit Baghdasaryan, and Eric Tiffany. *FIDO UAF Protocol Specification v1.0*. In collab. with Dirk Balfanz, Brad Hill, and Jeff Hodges. Dec. 8, 2014. URL: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html (visited on 03/22/2024).

[79] Haojun Liu et al. “Merkle Tree: A Fundamental Component of Blockchains”. In: *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*. 2021 International Conference on Electronic Information Engineering and Computer Science (EIECS). Sept. 2021, pp. 556–561. DOI: 10.1109/EIECS53707.2021.9588047. URL: https://ieeexplore.ieee.org/document/9588047 (visited on 12/07/2023).

[80] Ben Lynn. *Elliptic Curves - Explicit Addition Formulae*. 2015. URL: https://crypto.stanford.edu/pbc/notes/elliptic/explicit.html (visited on 11/09/2023).

[81] Mary Maller et al. *Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings*. Publication info: Preprint. MINOR revision. 2019. URL: https://eprint.iacr.org/2019/099 (visited on 02/02/2024).

[82] *merlin - Rust*. May 1, 2019. URL: https://docs.rs/merlin/latest/merlin/ (visited on 04/02/2024).

[83] Thibault Meunier. *Humanity wastes about 500 years per day on CAPTCHAs. It’s time to end this madness*. The Cloudflare Blog. May 13, 2021. URL: https://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood (visited on 03/22/2024).

[84] Jim Miller. *Coordinated disclosure of vulnerabilities affecting Girault, Bulletproofs, and PlonK*. Trail of Bits Blog. Apr. 13, 2022. URL: https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/ (visited on 11/03/2023).

[85] Jim Miller. *The Frozen Heart vulnerability in Bulletproofs*. Trail of Bits Blog. Apr. 15, 2022. URL: https://blog.trailofbits.com/2022/04/15/the-frozen-heart-vulnerability-in-bulletproofs/ (visited on 11/03/2023).

[86] Jim Miller. *The Frozen Heart vulnerability in PlonK*. Trail of Bits Blog. Apr. 18, 2022. URL: https://blog.trailofbits.com/2022/04/18/the-frozen-heart-vulnerability-in-plonk/ (visited on 11/03/2023).

[87] Arno Mittelbach and Marc Fischlin. "Pseudorandomness and Computational Indistinguishability". In: *The Theory of Hash Functions and Random Oracles: An Approach to Modern Cryptography*. Ed. by Arno Mittelbach and Marc Fischlin. Information Security and Cryptography. Cham: Springer International Publishing, 2021, pp. 95–159. ISBN: 978-3-030-63287-8. DOI: 10.1007/978-3-030-63287-8_3. URL: https://doi.org/10.1007/978-3-030-63287-8_3 (visited on 12/20/2023).

[88] *Moneropedia: Bulletproofs*. getmonero.org, The Monero Project. Dec. 5, 2018. URL: https://www.getmonero.org/resources/moneropedia/bulletproofs.html (visited on 04/09/2024).

[89] Dustin Moody et al. "Report on Pairing-based Cryptography". In: *NIST* 120 (2015) (Feb. 3, 2015). Last Modified: 2018-11-10T10:11-05:00 Publisher: Dustin Moody, Rene C. Peralta, Ray A. Perlner, Andrew R. Regenscheid, Allen L. Roginsky, Lidong Chen, pp. 11–27. URL: https://www.nist.gov/publications/report-pairing-based-cryptography (visited on 01/23/2024).

[90] Elena Nadolinski. "Demystifying Zero Knowledge Proofs \x5bFINAL\x5d". 2020. URL: https://docs.google.com/presentation/d/1gfB6WZMvM9mmDKofFibIgsyYShdf0RV_Y8TLz3k1Ls0 (visited on 04/05/2024).

[91] Yoav Nir and Simon Josefsson. *Curve25519 and Curve448 for the Internet Key Exchange Protocol Version 2 (IKEv2) Key Agreement*. Request for Comments RFC 8031. Num Pages: 8. Internet Engineering Task Force, Dec. 2016. DOI: 10.17487/RFC8031. URL: https://datatracker.ietf.org/doc/rfc8031 (visited on 04/09/2024).

[92] "NIST Removes Cryptography Algorithm from Random Number Generator Recommendations". In: *NIST* (Apr. 21, 2014). Last Modified: 2023-01-25T11:51-05:00. URL: https://www.nist.gov/news-events/news/2014/04/nist-removes-cryptography-algorithm-random-number-generator-recommendations (visited on 01/25/2024).

[93] *Non-interactive zero-knowledge proof*. In: *Wikipedia*. Page Version ID: 1211847071. Mar. 4, 2024. URL: https://en.wikipedia.org/w/index.php?title=Non-interactive_zero-knowledge_proof&oldid=1211847071 (visited on 04/05/2024).

[94] Bjorn Oude Roelink. *NIZKP-Benchmark: Benchmark for the zk-SNARK, zk-STARK, and Bulletproof non-interactive zero-knowledge proof (NIZKP) protocols*. Apr. 22, 2024. URL: https://github.com/bjornouderoelink/NIZKP-Benchmark (visited on 04/22/2024).

[95] Bjorn Oude Roelink, Mohammed El-Hajj, and Dipti Sarmah. "Systematic review: Comparing zk-SNARK, zk-STARK, and bulletproof protocols for privacy-preserving authentication". In: *SECURITY AND PRIVACY* n/a (n/a 2024). _eprint: https://onlinelibrary.wiley. e401. ISSN: 2475-6725. DOI: 10.1002/spy2.401. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.401 (visited on 04/18/2024).

[96] Andreea-Elena Panait and Ruxandra F. Olimid. "On Using zk-SNARKs and zk-STARKs in Blockchain-Based Identity Management". In: *Innovative Security Solutions for Information Technology and Communications*. Ed. by Diana Maimut, Andrei-George Oprina, and Damien Sauveron. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 130–145. ISBN: 978-3-030-69255-1. DOI: 10.1007/978-3-030-69255-1_9.

[97] Bryan Parno et al. *Pinocchio: Nearly Practical Verifiable Computation*. Publication info: Published elsewhere. This is the full version of the IEEE Symposium on Security & Privacy 2013 paper. 2013. URL: https://eprint.iacr.org/2013/279 (visited on 10/31/2023).

[98] Juha Partala, Tri Hong Nguyen, and Susanna Pirttikangas. "Non-Interactive Zero-Knowledge for Blockchain: A Survey". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 227945–227961. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3046025.

[99] Torben Pryds Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1992, pp. 129–140. ISBN: 978-3-540-46766-3. DOI: 10.1007/3-540-46766-1_9.

[100] Maksym Petkus. *Why and How zk-SNARK Works*. June 17, 2019. DOI: 10.48550/arXiv.1906.07221. arXiv: 1906.07221[cs,math]. URL: http://arxiv.org/abs/1906.07221 (visited on 11/07/2023).

[101] David Pointcheval and Jacques Stern. "Security Proofs for Signature Schemes". In: *Advances in Cryptology — EUROCRYPT '96*. Ed. by Ueli Maurer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 387–398. ISBN: 978-3-540-68339-1. DOI: 10.1007/3-540-68339-9_33.

[102] J. M. Pollard. "Monte Carlo Methods for Index Computation (mod p)". In: *Mathematics of Computation* 32.143 (July 1978), p. 918. ISSN: 00255718. DOI: 10.2307/2006496. URL: https://www.jstor.org/stable/2006496?origin=crossref (visited on 01/25/2024).

[103] *Quantum Computer and Simulator – Amazon Braket Pricing – AWS*. Amazon Web Services, Inc. 2024. URL: https://aws.amazon.com/braket/pricing/ (visited on 01/24/2024).

[104] *Ristretto - The Ristretto Group*. 2018. URL: https://ristretto.group/ristretto.html (visited on 04/02/2024).

[105] Martina Rossi et al. "Using Shor's algorithm on near term Quantum computers: a reduced version". In: *Quantum Machine Intelligence* 4.2 (July 2, 2022), p. 18. ISSN: 2524-4914. DOI: 10.1007/s42484-022-00072-2. URL: https://doi.org/10.1007/s42484-022-00072-2 (visited on 01/23/2024).

[106] *SafeCurves: CM field discriminants*. Oct. 29, 2013. URL: https://safecurves.cr.yp.to/disc.html (visited on 01/25/2024).

[107] Elvira Sánchez Ortiz. *Zero-Knowledge Proofs applied to finance*. Publisher: University of Twente. 2020. URL: https://essay.utwente.nl/83802/ (visited on 08/10/2023).

[108] Jörg Schwenk. "Cryptography: Integrity and Authenticity". In: *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*. Ed. by Jörg Schwenk. Information Security and Cryptography. Cham: Springer International Publishing, 2022, pp. 43–62. ISBN: 978-3-031-19439-9. DOI: 10.1007/978-3-031-19439-9_3. URL: https://doi.org/10.1007/978-3-031-19439-9_3 (visited on 12/20/2023).

[109] P.W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Proceedings 35th Annual Symposium on Foundations of Computer Science. Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[110] Ritesh Kumar Singh et al. “Energy Consumption Analysis of LPWAN Technologies and Lifetime Estimation for IoT Application”. In: *Sensors (Basel, Switzerland)* 20.17 (Aug. 25, 2020), p. 4794. ISSN: 1424-8220. DOI: 10.3390/s20174794. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7506725/ (visited on 04/09/2024).

[111] Unathi Skosana and Mark Tame. “Demonstration of Shor’s factoring algorithm for N $$=$$ 21 on IBM quantum processors”. In: *Scientific Reports* 11.1 (Aug. 16, 2021). Number: 1 Publisher: Nature Publishing Group, p. 16599. ISSN: 2045-2322. DOI: 10.1038/s41598-021-95973-w. URL: https://www.nature.com/articles/s41598-021-95973-w (visited on 01/23/2024).

[112] *STARK Technology*. StarkWare. 2024. URL: https://starkware.co/stark/ (visited on 04/29/2024).

[113] Josh Swihart, Benjamin Winston, and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. Electric Coin Company. Feb. 5, 2019. URL: https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/ (visited on 11/03/2023).

[114] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. *Rescue-Prime: a Standard Specification (SoK)*. Publication info: Preprint. MINOR revision. 2020. URL: https://eprint.iacr.org/2020/1143 (visited on 03/05/2024).

[115] Eric W. Weisstein. *Euler Formula*. Publisher: Wolfram Research, Inc. Dec. 12, 2023. URL: https://mathworld.wolfram.com/ (visited on 12/18/2023).

[116] Eric W. Weisstein. *Fast Fourier Transform*. Publisher: Wolfram Research, Inc. Nov. 3, 2023. URL: https://mathworld.wolfram.com/ (visited on 11/09/2023).

[117] Eric W. Weisstein. *Principal Root of Unity*. Publisher: Wolfram Research, Inc. Dec. 12, 2023. URL: https://mathworld.wolfram.com/ (visited on 12/18/2023).

[118] Eric W. Weisstein. *Root of Unity*. Publisher: Wolfram Research, Inc. Dec. 12, 2023. URL: https://mathworld.wolfram.com/ (visited on 12/18/2023).

[119] Tara Whalen et al. “Let The Right One In: Attestation as a Usable CAPTCHA Alternative”. In: (Aug. 8, 2022).

[120] Zooko Wilcox. “Privacy for Everyone”. Devcon4. Nov. 1, 2018. URL: https://slideslive.com/38911617/privacy-for-everyone?ref=og-meta-tags (visited on 04/05/2024).

[121] *winterfell - crates.io: Rust Package Registry*. Feb. 21, 2024. URL: https://crates.io/crates/winterfell/0.8.1 (visited on 03/15/2024).

[122] *Zcash’s Zero Knowledge Proofs, ZK Snarks, and More*. Gemini. Oct. 3, 2023. URL: https://www.gemini.com/cryptopedia/zcash-zero-knowledge-proof-zk-snarks-mining,%20https://www.gemini.com/cryptopedia/zcash-zero-knowledge-proof-zk-snarks-mining (visited on 04/15/2024).

[123] *Zero-knowledge rollups*. ethereum.org. Mar. 13, 2024. URL: https://ethereum.org/en/developers/docs/scaling/zk-rollups/ (visited on 04/15/2024).

[124] *zkcrypto/bellman*. original-date: 2015-12-24T10:00:37Z. Mar. 10, 2024. URL: https://github.com/zkcrypto/bellman (visited on 03/15/2024).

[125] *zkcrypto/bulletproofs*. original-date: 2021-01-14T21:17:17Z. Mar. 7, 2024. URL: https://github.com/zkcrypto/bulletproofs (visited on 03/15/2024).

[126] *zkp::toolbox::batch_verifier::BatchVerifier - Rust*. Dec. 6, 2019. URL: https://doc-internal.dalek.rs/zkp/toolbox/batch_verifier/struct.BatchVerifier.html (visited on 04/08/2024).

Appendix A

zk-SNARK

A.1 zk-SNARK protocol steps (Pinocchio with zero-knowledge)

In this section we will describe the Pinocchio zk-SNARK implementation by Parno et al. [97] with additional zero-knowledge as described in [100]. There are other implementations such as one named Groth16 after the 2016 paper by Groth [63]. For the detailed workings of these implementations we refer to the original papers.

A.1.1 Setup

- Select the generator

  $$g \tag{A.1}$$

  This must be a generator point on a curve in order to be able to leverage elliptic curves as described in subsection 4.2.5 for the cryptographic pairing of Equation A.2.

- Select the cryptographic pairing

  $$e \tag{A.2}$$

  This cryptographic pairing is used to deterministically map two (encrypted) inputs from one set of numbers to a different set of numbers as described in subsection 4.2.2.

- Use the QAP as described in subsection 4.2.6 to convert a function $f(u) = y$ with n total variables of which m are input/output variables into a polynomial form of size $n + 1$ and a degree d equal to the number of operations in the arithmetic circuit

  $$(\{l_i(x), r_i(x), o_i(x)\}_{i\in\{0,\ldots,n\}}, t(x)) \tag{A.3}$$

- Sample the random values

  $$s, \rho_l, \rho_r, \alpha_l, \alpha_r, \alpha_o, \beta, \gamma \tag{A.4}$$

  Here s is the used secret. The values $\alpha_l, \alpha_r, \alpha_o$ are used to ensure that the verification fails if the prover does not exponentiate secret s with the same value. We require three different α values in order to ensure that the prover cannot falsify a proof by re-using or swapping operand polynomials in Equation A.22, e.g. by using $L(s) \times L(s) = O(s)$ or $L(s) \times O(s) = R(s)$ instead of $L(s) \times R(s) = O(s)$. The values ρ_l, ρ_r, β are used to enforce usage of the same variable values v_i in each of the operand polynomials of Equation A.14, but in a more succinct manner. The value γ is used to prevent the prover from obtaining the encrypted β which ensures the soundness.

- Set

$$\rho_o = \rho_l \cdot \rho_r \tag{A.5}$$

This is the ρ value calculated for the output polynomial.

- Set the operand generators using Equation A.1 and Equation A.5

$$g_l = g^{\rho_l} \tag{A.6a}$$
$$g_r = g^{\rho_r} \tag{A.6b}$$
$$g_o = g^{\rho_o} \tag{A.6c}$$

Here we randomized the generator g for each operand polynomial to prevent the prover from being able to prove a different statement by adding an arbitrary value v' to the polynomial.

- Set the proving key

$$\begin{aligned} (\{g^{s^k}\}_{k\in[d]}, \\ \{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\}_{i\in\{0,\ldots,n\}}, \\ \{g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\}_{i\in\{m+1,\ldots,n\}}, \\ g_l^{t(s)}, g_r^{t(s)}, g_o^{t(s)}, g_l^{\alpha_l t(s)}, g_r^{\alpha_r t(s)}, g_o^{\alpha_o t(s)}, g_l^{\beta t(s)}, g_r^{\beta t(s)}, g_o^{\beta t(s)}) \end{aligned} \tag{A.7}$$

- Set the verification key using the outputs of Equation A.1, Equation A.6, Equation A.3, and Equation A.4

$$(g, g_o^{t(s)}, \{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\gamma}, g^{\beta_\gamma}) \tag{A.8}$$

A.1.2 Proving

- Compute values for all intermediary variables of the function $f(u)$ computation for input u

$$\{v_i\}_{i\in\{m+1,\ldots,n\}} \tag{A.9}$$

These intermediary variables are computed by the prover and exist to require certain input or output values in the polynomial. The verification will only succeed if the prover uses these values as input while generating the proof.

- Set the composite operand polynomials using Equation A.3 and Equation A.9

$$L(x) = l_0(x) + \sum_{i=1}^{n} v_i \cdot l_i(x) \tag{A.10a}$$
$$R(x) = r_0(x) + \sum_{i=1}^{n} v_i \cdot r_i(x) \tag{A.10b}$$
$$O(x) = o_0(x) + \sum_{i=1}^{n} v_i \cdot o_i(x) \tag{A.10c}$$

These composite operand polynomials are the composite of the operand variable polynomial for each distinct variable v_i. By using composite of multiple operand variable polynomials the value of each variable can be set separately to support multiple variables, while the value is enforced to be consistent when the same variable is used.

- Sample random values

$$\delta_l, \delta_r, \delta_o \tag{A.11}$$

These values are used to apply as a random δ-shift which makes the proof indistinguishable from random, and therefore zero-knowledge. We require a separate value for each operand polynomial to prevent information leakage.

- Find using Equation A.3, Equation A.10, and Equation A.11

$$h(x) = (L(x)R(x) - O(x))/(t(x)) + \delta_r L(x) + \delta_l R(x) + \delta_l \delta_r t(x) - \delta_o \tag{A.12}$$

This polynomial is calculated such that $p(x) = t(x) \cdot h(x)$.

- Assign the variable values to the encrypted polynomials of the prover and apply the zero-knowledge δ-shift; using Equation A.7, Equation A.9, and Equation A.11

$$g_l^{L_p(s)} = (g_l^{t(s)})^{\delta_l} \cdot \prod_{i=m+1}^{n} (g_l^{l_i(s)})^{v_i} \tag{A.13a}$$

$$g_r^{R_p(s)} = (g_r^{t(s)})^{\delta_r} \cdot \prod_{i=m+1}^{n} (g_r^{r_i(s)})^{v_i} \tag{A.13b}$$

$$g_o^{O_p(s)} = (g_o^{t(s)})^{\delta_o} \cdot \prod_{i=m+1}^{n} (g_o^{o_i(s)})^{v_i} \tag{A.13c}$$

- Assign the α-shifted pairs using Equation A.7, Equation A.9, and Equation A.11

$$g_l^{L'_p(s)} = (g_l^{\alpha_l t(s)})^{\delta_l} \cdot \prod_{i=m+1}^{n} (g_l^{\alpha_l l_i(s)})^{v_i} \tag{A.14a}$$

$$g_r^{R'_p(s)} = (g_r^{\alpha_r t(s)})^{\delta_r} \cdot \prod_{i=m+1}^{n} (g_r^{\alpha_r r_i(s)})^{v_i} \tag{A.14b}$$

$$g_o^{O'_p(s)} = (g_o^{\alpha_o t(s)})^{\delta_o} \cdot \prod_{i=m+1}^{n} (g_o^{\alpha_o o_i(s)})^{v_i} \tag{A.14c}$$

- Assign the consistency polynomial using Equation A.7, Equation A.9, and Equation A.11

$$g^{Z(s)} = (g_l^{\beta t(s)})^{\delta_l} (g_r^{\beta t(s)})^{\delta_r} (g_o^{\beta t(s)})^{\delta_o} \cdot \prod_{i=m+1}^{n} (g_l^{\beta l_i(s)} g_r^{\beta r_i(s)} g_o^{\beta o_i(s)})^{v_i} \tag{A.15}$$

- Set the proof from the assigned values in Equation A.1 together with Equation A.12, Equation A.13, Equation A.14, and Equation A.15

$$(g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g^{h(s)}, g_l^{L'_p(s)}, g_r^{R'_p(s)}, g_o^{O'_p(s)}, g^{Z(s)}) \tag{A.16}$$

A.1.3 Verifying

- Parse the proof from Equation A.16 as

$$(g_l^{L_p}, g_r^{R_p}, g_o^{O_p}, g^h, g_l^{L'_p}, g_r^{R'_p}, g_o^{O'_p}, g^Z) \tag{A.17}$$

- Compute values for all intermediary variables of the function $f(u)$ computation for input u

$$\{v_i\}_{i \in \{m+1, \ldots, n\}} \tag{A.18}$$

- Assign input and output values to the encrypted polynomials of the verifier and combine them using Equation A.8, and Equation A.18.

$$g_l^{L_v(s)} = g_l^{l_0(s)} \cdot \prod_{i=1}^{m} (g_l^{l_i(s)})^{v_i} \tag{A.19a}$$

$$g_r^{R_v(s)} = g_r^{r_0(s)} \cdot \prod_{i=1}^{m} (g_r^{r_i(s)})^{v_i} \tag{A.19b}$$

$$g_o^{O_v(s)} = g_o^{o_0(s)} \cdot \prod_{i=1}^{m} (g_o^{o_i(s)})^{v_i} \tag{A.19c}$$

- Check polynomial restrictions using Equation A.2, Equation A.8, and Equation A.17

$$e(g_l^{L_p}, g^{\alpha_l}) \stackrel{?}{=} e(g_l^{L'_p}, g) \tag{A.20a}$$

$$e(g_r^{R_p}, g^{\alpha_r}) \stackrel{?}{=} e(g_r^{R'_p}, g) \tag{A.20b}$$

$$e(g_o^{O_p}, g^{\alpha_o}) \stackrel{?}{=} e(g_o^{O'_p}, g) \tag{A.20c}$$

  This check ensures that the values used in Equation A.13 correspond to the values used in Equation A.14.

- Check values consistency using Equation A.2, Equation A.8, and Equation A.17

$$e(g_l^{L_p} g_r^{R_p} g_o^{O_p}, g^{\beta\gamma}) \stackrel{?}{=} e(g^Z, g^{\gamma}) \tag{A.21}$$

  This check ensures that the used values v_i are consistent operand variable polynomials in Equation A.13 and Equation A.14 using the consistency polynomial from Equation A.15.

- Check operations using Equation A.2, Equation A.8, and Equation A.17

$$e(g_l^{L_p} g_l^{L_v(s)}, g_r^{R_p} g_r^{R_v(s)}) \stackrel{?}{=} e(g_o^{t(s)}, g^h) \cdot e(g_o^{O_p} g_o^{O_v(s)}, g) \tag{A.22}$$

  This check ensures that the known polynomial evaluated on secret s is a multiple of the target polynomial.

- Accept the proof if and only if all checks pass.

Appendix B

Bulletproof

B.1 Bulletproof protocol steps for arithmetic circuits

In this section we describe the mathematical steps in the Bulletproof protocol according to Protocol 3 in the Bulletproof paper by Bünz et al. [33]. We adapt the protocol to make it non-interactive according to the steps from Section 4.4 of the same paper. Note: this protocol requires a trusted setup to generate a common reference string for $g, h, \mathbf{g}, \mathbf{h}$. This trusted setup can be avoided by generating the values using the secure hash and a seed value as described in Section 4.4 of the Bulletproof paper [33].

B.1.1 Inputs

$$\text{Secure hash function H} \tag{B.1a}$$
$$g, h \in \mathbb{G} \tag{B.1b}$$
$$\mathbf{g}, \mathbf{h} \in \mathbb{G}^n \tag{B.1c}$$
$$\mathbf{W}_L, \mathbf{W}_R, \mathbf{W}_O \in \mathbb{Z}_p^{Q\times n} \tag{B.1d}$$
$$\mathbf{W}_V \in \mathbb{Z}_p^{Q\times m} \tag{B.1e}$$
$$\mathbf{c} \in \mathbb{Z}_p^{Q} \tag{B.1f}$$
$$\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O \in \mathbb{Z}_p^{n} \tag{B.1g}$$
$$\gamma \in \mathbb{Z}_p^{m} \tag{B.1h}$$

B.1.2 Proving

- Generate random values

$$\alpha, \beta, \rho \in \mathbb{Z}_p \tag{B.2}$$

- Commit to $\mathbf{a}_L, \mathbf{a}_R$ using Equation B.1 and Equation B.2

$$A_I = h^{\alpha} \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R} \in \mathbb{G} \tag{B.3}$$

- Commit to $\mathbf{a}_O$ using Equation B.1 and Equation B.2

$$A_O = h^{\beta} \mathbf{g}^{\mathbf{a}_O} \in \mathbb{G} \tag{B.4}$$

- Generate random blinding vectors

$$\mathbf{s}_L, \mathbf{s}_R \in \mathbb{Z}_p^n \tag{B.5}$$

- Commit to $\mathbf{s}_L, \mathbf{s}_R$ using Equation B.1, Equation B.2 and Equation B.5

$$S = h^\rho \mathbf{g}^{\mathbf{s}_L} \mathbf{h}^{\mathbf{s}_R} \in \mathbb{G} \tag{B.6}$$

- Generate challenge y using the problem statement (description of a circuit) st and Equation B.1, Equation B.3, Equation B.4, and Equation B.6

$$y = \mathrm{H}(\mathrm{st}, A_I, A_O, S) \in \mathbb{Z}_p^* \tag{B.7}$$

- Generate challenge z using Equation B.1, Equation B.3, Equation B.4, Equation B.6, and Equation B.7

$$z = \mathrm{H}(A_I, A_O, S, y) \in \mathbb{Z}_p^* \tag{B.8}$$

- Compute per witness challenge using Equation B.7

$$\mathbf{y}^n = (1, y, y^2, \dots, y^{n-1}) \in \mathbb{Z}_p^n \tag{B.9}$$

- Compute per constraint challenge using Equation B.8

$$\mathbf{z}_{[1:]}^{Q+1} = (z, z^2, \dots, z^Q) \in \mathbb{Z}_p^Q \tag{B.10}$$

- Compute independent of witness using Equation B.1, Equation B.9, and Equation B.10

$$\delta(y, z) = \langle \mathbf{y}^{-n} \circ (\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_R), \mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_L \rangle \tag{B.11}$$

- Compute input polynomials using Equation B.1, Equation B.9, and Equation B.10

$$l(X) = \mathbf{a}_L \cdot X + \mathbf{a}_O \cdot X^2 + \mathbf{y}^{-n} \circ (\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_R) \cdot X + \mathbf{s}_L \cdot X^3 \in \mathbb{Z}_p^n[X] \tag{B.12a}$$

$$r(X) = \mathbf{y}^n \circ \mathbf{a}_R \cdot X - \mathbf{y}^n + \mathbf{z}_{[1:]}^{Q+1} \cdot (\mathbf{W}_L \cdot X + \mathbf{W}_O) + \mathbf{y}^n \circ \mathbf{s}_R \cdot X^3 \in \mathbb{Z}_p^n[X] \tag{B.12b}$$

- Compute target polynomial using Equation B.12

$$t(X) = \langle l(X), r(X) \rangle = \sum_{i=1}^{6} t_i \cdot X^i \in \mathbb{Z}_p[X] \tag{B.13}$$

- Compute using Equation B.1

$$\mathbf{w} = \mathbf{W}_L \cdot \mathbf{a}_L + \mathbf{W}_R \cdot \mathbf{a}_R + \mathbf{W}_O \cdot \mathbf{a}_O \tag{B.14}$$

- Compute using Equation B.1, Equation B.7, Equation B.8, and Equation B.11

$$t_2 = \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle - \langle \mathbf{a}_O, \mathbf{y}^n \rangle + \langle \mathbf{z}_{[1:]}^{Q+1}, \mathbf{w} \rangle + \delta(y, z) \in \mathbb{Z}_p \tag{B.15}$$

- Generate random values

$$\tau_i \in \mathbb{Z}_p \quad \forall_i \in [1,3,4,5,6] \tag{B.16}$$

- Compute commitments to t_i (except t_2) using Equation B.13, and Equation B.16

$$T_i = g^{t_i} h^{\tau_i} \quad \forall_i \in [1,3,4,5,6] \tag{B.17}$$

- Generate challenge x using Equation B.1, Equation B.3, Equation B.4, Equation B.6, and Equation B.8

$$x = \mathrm{H}(A_I, A_O, S, z) \in \mathbb{Z}_p^* \tag{B.18}$$

- Compute using Equation B.12

$$\mathbf{l} = l(x) \in \mathbb{Z}_p^n \tag{B.19a}$$
$$\mathbf{r} = r(x) \in \mathbb{Z}_p^n \tag{B.19b}$$

- Compute using Equation B.19

$$\hat{t} = \langle \mathbf{l}, \mathbf{r} \rangle \in \mathbb{Z}_p \tag{B.20}$$

- Compute blinding value for $\hat{t}$ using Equation B.1, Equation B.10, and Equation B.16

$$\tau_x = \sum_{i=1, i\neq 2}^{6} \tau_i \cdot x^i + x^2 \cdot \langle \mathbf{z}_{[1:]}^{Q+1}, \mathbf{W}_V \cdot \gamma \rangle \in \mathbb{Z}_p \tag{B.21}$$

- Compute blinding value μ for P using Equation B.2

$$\mu = \alpha \cdot x + \beta \cdot x^2 + \rho \cdot x^3 \in \mathbb{Z}_p \tag{B.22}$$

- Set the proof as the output of Equation B.3, Equation B.4, Equation B.6, Equation B.17, Equation B.19, Equation B.20, Equation B.21, and Equation B.22.

$$A_I, A_O, S, T_1, T_3, T_4, T_5, T_6, \tau_x, \mu, \hat{t}, \mathbf{l}, \mathbf{r} \tag{B.23}$$

B.1.3 Verifying

- Parse the proof from Equation B.23

$$A_I, A_O, S, T_1, T_3, T_4, T_5, T_6, \tau_x, \mu, \hat{t}, \mathbf{l}, \mathbf{r} \tag{B.24}$$

- Generate challenge y using the problem statement (description of a circuit) st and Equation B.24

$$y = \mathrm{H}(\mathrm{st}, A_I, A_O, S) \in \mathbb{Z}_p^* \tag{B.25}$$

- Generate challenge z using Equation B.24 and Equation B.25

$$z = \mathrm{H}(A_I, A_O, S, y) \in \mathbb{Z}_p^* \tag{B.26}$$

- Compute per witness challenge using Equation B.25

$$\mathbf{y}^n = (1, y, y^2, \dots, y^{n-1}) \in \mathbb{Z}_p^n \tag{B.27}$$

- Compute per constraint challenge using Equation B.26

$$\mathbf{z}_{[1:]}^{Q+1} = (z, z^2, \dots, z^Q) \in \mathbb{Z}_p^Q \tag{B.28}$$

- Compute independent of witness using Equation B.1, Equation B.27, and Equation B.28

$$\delta(y, z) = \langle \mathbf{y}^{-n} \circ (\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_R), \mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_L \rangle \tag{B.29}$$

- Generate challenge x using Equation B.24 and Equation B.26

$$x = \mathrm{H}(A_I, A_O, S, z) \in \mathbb{Z}_p^* \tag{B.30}$$

- Compute using Equation B.25

$$h'_i = h_i^{y^{-i+1}} \quad \forall i \in [1, n] \tag{B.31a}$$

$$\mathbf{h}' = (h_1, h_2^{y^{-1}}, \dots, h_n^{y^{-n+1}}) \tag{B.31b}$$

- Compute the weights for $\mathbf{a}_L$, $\mathbf{a}_R$, and $\mathbf{a}_O$ using Equation B.1, Equation B.27, and Equation B.28

$$W_L = \mathbf{h}'^{\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_L} \tag{B.32a}$$

$$W_R = \mathbf{g}^{\mathbf{y}^{-n} \circ (\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_R)} \tag{B.32b}$$

$$W_O = \mathbf{h}'^{\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_O} \tag{B.32c}$$

- Compute the commitment to $l(x), r(x)$ using Equation B.24, Equation B.31, and Equation B.32

$$P = A_I^x \cdot A_O^{(x^2)} \cdot \mathbf{h}'^{-\mathbf{y}^n} \cdot W_L^x \cdot W_R^x \cdot W_O^x \cdot S^{(x^3)} \tag{B.33}$$

- Check that $\hat{t}$ is correct using Equation B.24

$$\hat{t} \overset{?}{=} \langle \mathbf{l}, \mathbf{r} \rangle \tag{B.34}$$

- Check that $\hat{t} = t(x)$ using Equation B.1, Equation B.24, Equation B.28, and Equation B.29

$$g^{\hat{t}} h^{\tau_x} \overset{?}{=} g^{x^2 \cdot (\delta(y,z) + \langle \mathbf{z}_{[1:]}^{Q+1}, \mathbf{c} \rangle)} \cdot \mathbf{V}^{x^2 \cdot (\mathbf{z}_{[1:]}^{Q+1} \cdot \mathbf{W}_V)} \cdot T_1^x \cdot \prod_{i=3}^{6} T_i^{(x^i)} \tag{B.35}$$

- Check that $\mathbf{l} = l(x)$ and $\mathbf{r} = r(x)$ using Equation B.1, Equation B.24, Equation B.31, and Equation B.33

$$P \overset{?}{=} h^{\mu} \cdot \mathbf{g}^{\mathbf{l}} \cdot \mathbf{h}'^{\mathbf{r}} \tag{B.36}$$

- Accept the proof if and only if all checks pass.