ZPiE: Zero-knowledge Proofs in Embedded systems

Xavier Salleras[1] and Vanesa Daza[1,2]

[1]Department of Information and Communication Technologies,
Universitat Pompeu Fabra, Barcelona, Spain
[2]CYBERCAT - Center for Cybersecurity Research of Catalonia
{xavier.salleras, vanesa.daza}@upf.edu

Abstract

Zero-Knowledge Proofs (ZKPs) are cryptographic primitives allowing a party to prove to another party that the former knows some information while keeping it secret. Such a premise can lead to the development of numerous privacy-preserving protocols in different scenarios, like proving knowledge of some credentials to a server without leaking the identity of the user. Even when the applications of ZKPs were endless, they were not exploited in the wild for a couple of decades due to the fact that computing and verifying proofs was too computationally expensive. However, the advent of efficient schemes (in particular, zk-SNARKs) made this primitive to break into the scene in fields like cryptocurrencies, smart-contracts, and more recently, self-sovereign scenarios: private-by-design identity management and authentication. Nevertheless, its adoption in environments like the Internet of Things (IoT) remains unexplored due to the computational limitations of embedded systems. In this paper, we introduce ZPiE, a C library intended to create ZKP applications to be executed in embedded systems. Its main feature is portability: it can be compiled, executed, and used out-of-the-box in a wide variety of devices. Moreover, our proof-of-concept has been proved to work smoothly in different devices with limited resources, which can execute state-of-the-art ZKP authentication protocols.

Keywords: Zero-Knowledge Proofs; SNARKs; Embedded Systems; Applied Cryptography.

This work has been accepted to be published in the special issue *Recent Advances in Security, Privacy, and Applied Cryptography* of the journal *Mathematics* (2021).

Contents

1 Introduction

The inclusion of 5G communications [1] involves the deployment of *network slices*, dedicated physical and logical networks for different use cases such as the Internet of Things (IoT) [2]. In this scenario, 5G grants a large bandwidth and a low latency to be able to handle a large density of IoT devices. This fact has a big impact on the evolution of the smart-cities paradigm, where the amount of IoT devices is growing exponentially [3]. In that regard, exposure to cyberattacks like identity spoofing [4] is a big concern. To prevent this specific issue, several solutions have been designed, for instance, using physical unclonable functions [5], which has been proved to be a secure way to identify IoT devices. However, such authentication mechanisms cannot be completed with a single-message protocol, which introduces some latency to the scheme. Other solutions like in [6] present a way to authenticate IoT devices using Blockchains [7], distributed networks sharing a unique ledger containing transactions made by different nodes in the network. This builds a decentralized way to authenticate devices, which is also lightweight and applicable to a large number of scenarios.

However, even when most authentication mechanisms are secure against spoofing attacks, users' privacy is still an issue to be addressed. Examples of this are medical devices exchanging patients' private information, or autonomous cars sharing their location with a server. The common point between all these devices is that they need to share sensitive data over the network, data which should not be traced by any Internet Service Provider (ISP) or eavesdropper. Self-sovereign identity systems [8] become an essential feature to implement in this scenario: systems where users can control, access, and transparently consent their identities, preventing entities to track and collect their data. Research integrating such features into IoT environments has been conducted, like in [9], where the authors combine self-sovereign identity systems with distributed ledger technologies. The main elements required to get these features are Zero-Knowledge Proofs (ZKPs).

ZKPs are cryptographic primitives gaining a lot of momentum thanks to technologies such as Blockchain. While cryptocurrencies like Bitcoin [10] allow any node to review the transactions, knowing the sender, receiver, and the amount of exchanged money (among all the information involving the transaction), cryptocurrencies such as Zcash [11] use a specific ZKP scheme that allows its users to make anonymous payments. ZKPs allow a party (the prover) to prove to another party (the verifier) that he knows some information while keeping it secret. In other words, the sender, the receiver, and the amount of exchanged money is kept secret while everyone in the network agrees on the correctness of the transaction.

1.1 Motivation

Beyond cryptocurrencies, ZKPs are also used in protocols such as `SANS`[12], a privacy-preserving authentication scheme where users prove their right to access a service, based on some previously granted attributes. On the other hand, blockchains like Ethereum [13] provide an architecture capable of executing programs *on-chain*. These programs, called *smart-contracts*, can be combined with ZKP schemes to grant more privacy to its users: for instance, issuing a payment after using some service, while keeping the identity of the user secret. Protocols based on such approaches would be desirable in IoT scenarios, where many services are used, and most of them (such as medical applications) collect sensitive data. However, ZKP schemes require high computational resources, especially when the proof is generated. Even when current ZKP implementations do a great job in terms of efficiency, they are far from being portable, especially for embedded systems, as most of them focus on web applications, with the usage of programming languages like JavaScript, Webassembly, or Python. Some solutions [14] try to distribute the operations to be done to generate the proof between trusted servers. However, such approaches require a trusted environment and a persistent and fast connection with the server. Optimal solutions would be novel implementations focused on devices with low resources.

1.2 Contributions

We introduce `ZPiE`, a portable C library for generating and verifying ZKPs. As an initial proof-of-concept, `ZPiE` implements the specific ZKP construction called zk-SNARK. Our library provides a clear API to create proofs, and to verify them. Upon doing several tests on different devices, we have proved that, unlike other state-of-the-art solutions, `ZPiE` can be executed in `x86`, `x86_64`, `aarch64` and `arm32` CPU architectures out-of-the-box. In addition, after performing several experiments using a `x86_64` CPU (compatible with other state-of-the-art libraries) and comparing the results with other solutions, we have proved that `ZPiE` has similar performance. Furthermore, our solution can be easily integrated with existing implementations to be used in smart-contracts.

1.3 Roadmap

In Section 2 we survey the relevant use cases and libraries developed to work with ZKPs. In Section 3 we expose the building blocks required to understand the primitive we are implementing. In Section 4 we present `ZPiE`

with all details. In Section 5 we include the performed experiments and their results. Conclusions and future work are provided in Section 6.

2 Related Work

In this section, we first introduce the main use cases of ZKPs. Later, we explain the different ZKP schemes and libraries used in real scenarios.

2.1 ZKP use cases

One of the main uses of ZKPs is enhancing privacy in cryptocurrencies. Along with the previously stated Zcash, other cryptocurrencies like Monero[0] use ZKPs to provide anonymous transactions when sending and receiving money over its network. Far from being limited to cryptocurrencies, ZKPs are used in Self-Sovereign Identity (SSI) systems as well, where users can control, consent, and widely use their identities among different services, along with other properties. A recent example of this is the solution introduced in [15], where authors introduce a Blockchain-based protocol combined with ZKPs, for enabling privacy-preserving authentication while removing the need for a central authority. They apply such an approach in electric vehicles scenarios, specifically when the vehicles need to charge the battery, and identification is requested by a charging provider.

Another recent research topic is introduced in `SANS` [12], where ZKPs could be used to authenticate users in scenarios like in 5G services. Such a protocol works as follows: upon signing up for using a 5G service, the service provider grants the user a digital signature signed by him, along with other parameters. Later, each time the user wants to use the service, he can do it anonymously: he computes a ZKP to demonstrate that he possesses a valid signature computed by the service provider, without leaking any other information. The service provider can validate the proof and grant the service without learning the identity of the user. Moreover, such an approach enhances privacy in distributed applications (DApps) as well. DApps are blockchain applications built using smart-contracts, which are binaries executed *on-chain*. For instance in Ethereum, such binaries are the result of compiling Solidity code[1]. Like this, a program that verifies proofs could be executed *on-chain*. This technology can grant more privacy to users using different kinds of DApps, not solely using ZKPs but also using approaches like the one introduced in [16]. However, as stated, even when the verification is executed *on-chain*, the proof still needs to be computed by the proving device. As such, it is still hard to use DApps based on ZKPs on IoT devices due to its hardware limitations.

2.2 ZKP schemes

Currently, there are several ZKP constructions used in the wild, each with its advantages and drawbacks. As shown in Table 1, one of the first efficient schemes was a zk-SNARK introduced in BSCTV'13 [17]. Such a scheme was later improved by the zk-SNARK introduced in Groth'16 [18], which is still one of the most efficient zk-SNARK schemes, especially when it comes to the verification algorithm. One of the main drawbacks of this kind of construction is the need for a trusted party that performs a *trusted setup*: a phase where some public parameters are generated, which will be used to generate and verify proofs. In this regard, Sonic [19] is a zk-SNARK that introduces a scheme where the setup can be updated for different circuits without the need to repeat the trusted setup generation. Another efficient zk-SNARK is Libra [20], which prover is guaranteed to outperform, even when the verifier does not have constant complexity. The most recent work on this topic was done by the authors in [21]. They introduce PlonK, a scheme with the same advantages that Sonic has, but improving the speed of the prover. Finally, recent research such as [22] shows a way to design a more efficient prover, while not compromising the verifier.

Table 1: Comparison of different ZKP constructions, in regards to their asymptotic efficiency, where n is the number of gates of the circuit and d its depth. (TS = trusted setup, PQS = post-quantum secure, $|\pi|$ = proof size, $\odot$ = per-statement, $\triangle$ = updatable)

Scheme	TS	PQS	Prove	Verify	$\lvert\pi\rvert$	Security Assumption
BSCTV'13 [17]	$\odot$	no	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE
Groth'16 [18]	$\odot$	no	$O(n \log n)$	$O(1)$	$O(1)$	q-PKE
Sonic [19]	$\triangle$	no	$O(n \log n)$	$O(1)$	$O(1)$	AGM
Libra [20]	$\triangle$	no	$O(n)$	$O(d \log n)$	$O(d \log n)$	q-SBDH, q-PKE
Bulletproofs [23]	no	no	$O(n)$	$O(n)$	$O(\log n)$	DLP
zk-STARKs [24]	no	yes	$O(n \log^2 n)$	$O(\log^2 n)$	$O(\log^2 n)$	CRHF

[0] https://www.getmonero.org/
[1] https://solidity.readthedocs.io/en/v0.5.3/

On the other hand, and beyond zk-SNARKs, we can find other ZKP schemes like Bulletproofs [23], which have the main advantage of not requiring a trusted setup. They are especially useful when the prover needs to compute a *range proof* [25], instead of an arithmetic circuit. Moreover, other concerns like post-quantum security have also arisen in the Zero-Knowledge field, and in this regard we have a scheme supposed to be post-quantum secure, called zk-STARKs (Zero-Knowledge Succinct Transparent ARgument of Knowledge) [24]. Post-quantum security is not a property of zk-SNARKs or Bulletproofs.

Finally, the soundness property of each of the schemes described relies on different security assumptions [26]. Most of the zk-SNARK constructions use a strong assumption called q-Power Knowledge of Exponent (q-PKE), which is not the best solution. On the other hand, Bulletproofs or zk-STARKs use better assumptions: the Discrete Logarithm Problem (DLP) and Collision Resistant Hash Functions (CRHF), respectively.

To develop ZKP applications, libraries like the one provided in this paper are required. One of the main libraries to accomplish this purpose is **libsnark**[2], a C++ library for constructing zk-SNARKs, which was used for some time by Zcash [11], based on the specific zk-SNARK construction introduced in [17], but supporting [18] as well, among others. Even when this library provides excellent benchmarks, one of the main drawbacks of this library, as the authors state, is not being well-optimized for ARM architectures.

Another library with similar benchmarks is **bellman**[3], implemented in Rust and meant for constructing zk-SNARKs, developed and currently used by Zcash.

Moreover, when it comes to developing DApps for the Ethereum blockchain, we previously stated that a verifier coded in Solidity is required. **ZoKrates**[4] is a python toolbox for zk-SNARKs intended to generate Solidity verifiers, to be deployed into the Ethereum blockchain. Furthermore, a similar approach is **snarkjs**[5], a JavaScript library for constructing zk-SNARKs. It includes a clear API for generating trusted setups using a fairly secure MPC protocol, for generating proofs, and for verifying them. Plus, it also provides an easy way to export the verifier in Solidity, to deploy it into the Ethereum blockchain.

3 Building Blocks: zk-SNARKs

In this section, we introduce the required building blocks in regards to our solution. We do a high-level overview of how to construct a zk-SNARK (based on the construction introduced in [18]), which is also the scheme used in our proof-of-concept.

3.1 Preliminaries

A Zero-Knowledge Proof (ZKP) [27] is a cryptographic primitive which allows a prover P to convince a verifier V that a statement is true, without leaking any secret information. A statement is a set of elements known by both parties, defined as $u \in \mathcal{L}$, where $\mathcal{L}$ is a Language. We have a witness w (the secret information only known by P) for a statement u if $(u, w) \in \mathcal{R}$, where $\mathcal{R}$ is a polynomial-time decidable binary relation associated with $\mathcal{L}$. Formally speaking, ZKPs must satisfy 3 properties:

- **Completeness:** If the statement is true, P must be able to convince V.
- **Soundness:** If the statement is false, P must not be able to convince V that the statement is true, except with negligible probability.
- **Zero-knowledge:** V must not learn any information from the proof beyond the fact that the statement is true.

Moreover, V might request a *proof of knowledge*, a property to guarantee that P knows a witness w. Such a witness is a vector of elements for which a set of operations hold. These operations are defined by a graph composed of different wires and gates called *circuit*. Such a circuit leads to a set of equations representing the inputs and the outputs of these gates. The equations are also called *constraints*.

Even when first schemes required P and V to interact several times, Non-Interactive ZKPs (NIZKPs) [28] emerged, allowing P to prove statements to V by sending him a single message.

zk-SNARKs are Zero-Knowledge Succinct and Non-interactive ARguments of Knowledge [17]. They are the most used ZKPs, because they are short and succinct: the proofs can be verified in a few milliseconds. However, they require a trusted setup where some public parameters are generated. These parameters, called the Common Reference String (CRS), are used either by P and V to generate and verify proofs. In the process of generating the CRS, a secret randomness τ is used, and such a randmoness should be destroyed afterwards. If an attacker gets τ, the soundness property of the scheme breaks: the attacker would be able to compute

[2]https://github.com/scipr-lab/libsnark
[3]https://github.com/zkcrypto/bellman/
[4]https://github.com/Zokrates/ZoKrates
[5]https://github.com/iden3/snarkjs

false proofs that anyone could verify as if they were correct. As such, the CRS is commonly computed using a secure Multi-Party Computation (MPC) protocol [29], where τ can only be leaked if all the participants are malicious. Therefore, zk-SNARKs are composed of three main algorithms: *setup*, *prove* and *verify*. The computing complexity of these algorithms depends on the number of operations that we do in the circuit, which is also the the number of gates n.

Regarding the security of zk-SNARKs, it mainly relies on the security of elliptic curves. Breaking the security of the elliptic curve used by a specific construction would lead to being able to generate false proofs and thus, breaking the soundness property of the scheme. Among the most used curves in ZKPs we have a Barreto-Naehrig curve [30] called BN128, which security level in practice is estimated to be 110-bits [31]. Another common curve in this scenario is BLS12-381 [11], which has around 128-bits of security, with the drawback of heavier group operations. More recent research is introduced in [32], where a new curve called BW6-761 is introduced. As stated by its authors, verification of proofs is at least five times faster than other state-of-the-art curves.

The zk-SNARK construction we will work on requires a pairing-friendly elliptic curve E over a finite field $\mathbb{F}_q$, where q is a prime number, with the bilinear groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order r and a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ being a bilinear map. We need the following generators: an element g being a generator for $\mathbb{G}_1$, an element h being a generator for $\mathbb{G}_2$, and $e(g, h)$ being a generator for $\mathbb{G}_T$. In order to represent group elements, we use the widely used additive notation: we write $[a]_1$ for g^a, $[b]_2$ for g^b, and $[c]_T$ for $e(g, h)^c$.

Arithmetic circuit. A circuit is a directed acyclic graph composed of different wires and gates, which lead to a set of equations relating the inputs and the outputs of these gates. The inputs and the output of this circuit, as well as the operations defined in the gates, are elements over a prime field $\mathbb{F}_r$, where r is the order of E.

Rank 1 Constraint System. A Rank 1 Constraint System (R1CS) is a system of equations which checks the correctness of all the operations in our circuit, by grouping them in *constraints*. Each constraint is composed of a multiplicative gate with its two inputs and its output. Having a statement u and a witness w such that $(u, w) \in \mathcal{R}$, a R1CS is defined as a set of vectors (a, b, c) defining our circuit, whose solution is a vector $s = (u, w)$ such that the following equation is satisfied:

$$< a, s > \cdot < b, s > - < c, s >= 0 \tag{1}$$

Quadratic Arithmetic Program. A Quadratic Arithmetic Program (QAP) is a polynomial representation of the R1CS, which bundles all its constraints into one. It is a tuple of the polynomials $(A, B, C, Z(x))$, where $Z(x)$ divides $A_i(x) \cdot B_i(x) - C_i(x)$ without remainder. They satisfy the following equation:

$$\sum_{i=0}^{m} s_i A_i(x)) \cdot \sum_{i=0}^{m} s_i B_i(x) - \sum_{i=0}^{m} s_i C_i(x) = H(x)Z(x) \tag{2}$$

3.2 Protocol

Let $\mathcal{R}$ be a relation composed of the elliptic curve E over $\mathbb{F}_q$, the pairing e and a QAP $(A, B, C, Z(x))$ representing a circuit. The Groth'16 construction is divided into three algorithms (further details can be found in Appendix A), as depicted in Figure 1:

- $pk, vk \leftarrow \mathit{Setup}(\mathcal{R})$: given the relation $\mathcal{R}$, the first step of the protocol generates a common reference string (CRS) $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$. From the CRS, some elements will be extracted into what we call the proving key pk, sent to the prover P to generate proofs. Moreover, other required elements will be taken into the verifying key vk, and sent to the verifier V to verify the proofs generated by P. In order to generate the CRS (i.e. pk and vk), a random set of values τ is used. This τ, also known as *trapdoor*, should be destroyed after performing the setup, as any party having τ would be able to generate false proofs. To solve this last drawback, the setup must be generated by a trusted party (i.e. a set of entities performing a secure MPC protocol). In scenarios such as cryptocurrencies using zk-SNARKs (i.e. Zcash), an untrusty setup could lead to malicious parties using τ to create false transactions, thus leading to losses of money.
- $\pi \leftarrow \mathit{Prove}(\mathcal{R}, pk, u, w)$: the prover generates a proof $\pi = ([\pi_A]_1, [\pi_B]_2, [\pi_C]_1)$, by multiplying u and w by some polynomials provided in σ. The prover also needs to compute the coefficients h of $H(x)$, which can be achieved in $O(n \log n)$ using Fast Fourier Transform (FFT) techniques, as explained in detail in Appendix B. Then, h is multiplied by a polynomial provided in σ. The number of multi-exponentiations required to compute π are (note that multi-exponentiations in $\mathbb{G}_2$ are more expensive than multi-exponentiations in $\mathbb{G}_1$):
  - to compute $[\pi_A]_1$: $|u| + |w|$ multi-exponentiations in $\mathbb{G}_1$.
  - to compute $[\pi_B]_2$: $|u| + |w|$ multi-exponentiations in $\mathbb{G}_2$.

– to compute $[\pi_C]_1$: $|u| + 2 \cdot |w| + |h|$ multi-exponentiations in $\mathbb{G}_1$.

- $0/1 \leftarrow Verify(\mathcal{R}, vk, u, \pi)$: the verifier accepts the proof (1) if an equation composed of three pairings [33] holds. Otherwise, the proof is rejected (0). Moreover, modifying a single bit of the proof leads to a proof that cannot be verified.

As such, the workflow of these algorithms in real applications would work as follows: we first need to compute the setup by means of a trusted party (step 1 in Figure 1). Later, this party shall share the required values with each of the involved parties, the prover and the verifier (step 2 in Figure 1). As shown in the steps 3 and 4, the prover computes the proof π and sends it to the verifier, who verifies it (step 5 in Figure 1). Steps 1 and 2 are performed only once, and later, the prover can compute as many proofs as he wants using the same values computed during the setup.

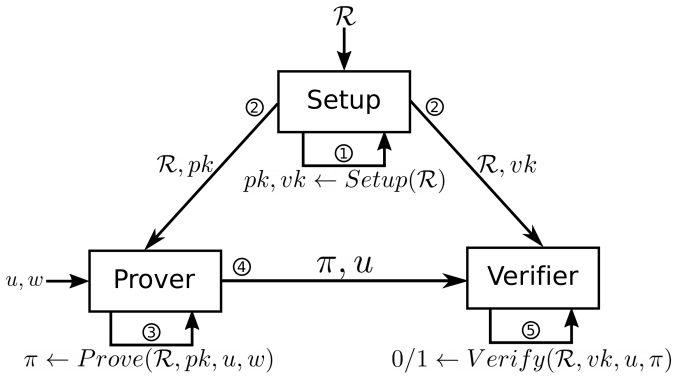

Figure 1: Zero-Knowledge Proof System.

4 Our Solution: ZPiE

In this section, we provide all details regarding `ZPiE`. We start with a comprehensive explanation about how it was designed, to later move to the technical details. We later provide the techniques used to improve the efficiency of the code. Finally, we provide an explanation on how to use the library.

4.1 Design

Our solution[6] has been designed to provide a clear interface for developing ZKP applications. As such, it provides an API to design circuits, perform the setup phase, generate proofs, and verify them. Furthermore, our solution can easily integrate the verifier in Solidity applications (i.e. using ZoKrates or snarkjs), in order to deploy smart-contracts into the Ethereum network. `ZPiE` can be used in devices with very low resources like a Raspberry Pi Zero, and in a wide variety of CPU architectures: `x86`, `x86_64`, `aarch64`, and `arm32`.

4.2 Implementation

Our solution has been designed with portability and scalability in mind. For such a reason, we decided to use C to code it, which is still widely used in embedded systems, and allows us full control over the hardware resources. For dealing with big numbers we use GNU GMP[7], which is one of the fastest approaches to do operations with large numbers in C. For the group operations over elliptic curves, and pairings, we rely on the MCL library[8], a well-optimized set of functions that offer us support for all the operations we need to perform. Both GMP and MCL offer support for `x86`, `x86_64`, `aarch64`, and `arm32` CPU architectures. Moreover, all the code has been designed to split the workload into threads, to increase the performance especially when the prover is executed in a multicore CPU. Regarding the circuit design, either the circuit parser and the circuit' code developed by the user are coded in pure C, so they are both compiled altogether along with all the other code for maximum performance.

[6] https://github.com/xevisalle/zpie
[7] https://gmplib.org/
[8] https://github.com/herumi/mcl

4.3 Efficiency

The first step to being able to use our proof system is to generate the CRS through the setup algorithm. Using Groth'16, the setup has a complexity of $O(n)$, so we need to compute a number of elements that depends only on the size of the circuit. Regarding the verifier, where the complexity time is constant ($O(1)$), he only needs to compute 3 pairings and verify that an equation holds, which is not expensive in terms of power consumption. Plus, the operations which require more power consumption are done by the prover when computing the proof. That is computing the h coefficients, and doing the multi-exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$.

4.3.1 Computing h coefficients

As stated in Appendix B, we rely on a FFT function to compute the coefficients in $O(n \log n)$. Our FFT function has been designed to be as efficient as possible. The size of the domain used for the FFTs in $N_e = 2^l$, where l is a big integer, so $|h| = N_e$. In total, the prover has to perform 3 inverse FFTs over a domain S, 3 FFTs over a shifted domain T, and one last inverse FFT over T. As depicted in Figure 2, this set of operations is, for any number of constraints, the same small percentage of all the operations performed to generate the proof.

4.3.2 Multi-exponentiations

Our solution uses three different multi-exponentiation approaches: the naive multi-exponentiation (which is a serial approach), the multi-exponentiation function provided MCL, and a Bos-Coster multi-exponentiation algorithm (further details provided in Appendix C). While the former achieves the worst results, MCL gets better marks. However, our Bos-Coster implementation achieves the best results. We split the Bos-Coster operations into chunks to increase the performance when using multithreading. As depicted in Figure 2, the multi-exponentiations represent a huge percentage of the operations to be done. In addition, the heap sorting, a step required after each Bos-Coster execution, increases exponentially in terms of global percentage.

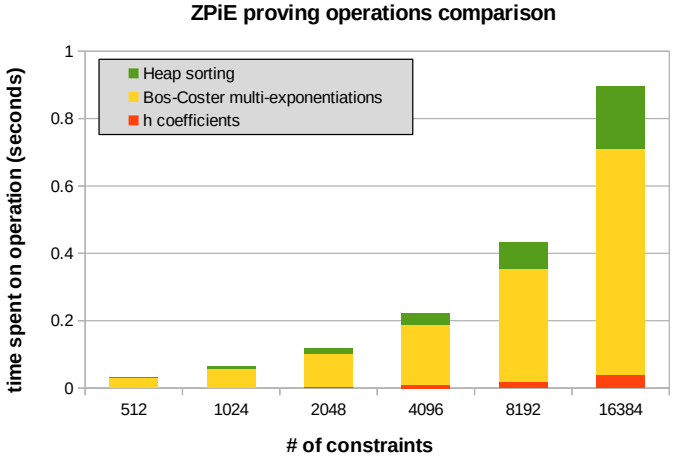

Figure 2: CPUs proving operations workload comparison of ZPiE executed in a i7-11370H CPU in single-thread mode. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

4.4 Applications

Some of the use cases of ZKPs involve proving to another party that we know the preimage of a hashed value, or that we have a valid signature for a secret message. We implemented both approaches using `ZPiE`, and provide an API to easily use them.

To use our library, we need to write a circuit which will be parsed later. To do so, Listing 1 shows an example on how to compute the MiMC [34] hash of a preimage x_in. As can be seen, we set a public output h, the hash of the secret preimage x_in (in such a process some randomness k is used as well). In other words, this circuit will compute a proof that, once verified, the verifier will be sure that who computed the proof knows x_in. As can be seen in Listing 2, `ZPiE` can compute the setup, the proof, and verify it by simply executing each algorithm as shown in the snippet.

In addition, and as shown in Listing 3, `ZPiE` can easily verify an EdDSA [35] signature by calling the function `verify_eddsa(...)`, providing the required parameters as shown in the snippet. In this scenario, the prover is proving to the verifier that he knows a valid signature for some secret message. In such regard, the code can be modified depending on the use case, to select which values are public or secret.

```c
#include "../circuits/mimc.c"

// main function called by the circuit parser
void circuit()
{
    element h, x_in, k;

    // we init the public output h, and private inputs x_in and k
    init_public(&h);
    init(&x_in);
    init(&k);

    // we manually set preimage and randomness values
    input(&x_in, "1234");
    input(&k, "112233445566");

    // compute a MiMC hash
    mimc7(&h, &x_in, &k);
}
```

Listing 1: Circuit example for computing the MiMC hash of a preimage.

```c
#include "../src/zpie.h"

int main()
{
    // we perform the setup (../data/provingkey.params and
    // ../data/verifyingkey.params)
    init_setup();
    perform_setup();

    // we generate a proof (../data/proof.params)
    init_prover();
    generate_proof();

    // we verify the proof (../data/proof.params)
    init_verifier();
    if (verify_proof()) printf("Proof verified.\n");
    else printf("Proof cannot be verified.\n");
}
```

Listing 2: Program execution example

```c
#include "../circuits/eddsa.c"

// main function called by the circuit parser
void circuit()
{
    element out[4];
    // we init the public output
    for (int i = 0; i < 4; ++i)
    {
        init_public(&out[i]);
    }

    // we provide some example values
    char *B1 = "52996192406415512816348655835182970302
    828744721907728940865211444827210015 53";
    char *B2 = "16950150798460657717958625567821834550
    301663161624707787222815936182638968203";
    char *R1 = "12629481114452250573734381948187634057
    004574874295483714632143261903118958 64";
    char *R2 = "12533500305127747239777484416561675628
    195562065959201739446841668623540883587";
    char *A1 = "21629779320182474195265732521833299809
    982444552305142529409236301104997786342";
    char *A2 = "90118124453810306641426220662183318451
    408818470349341666308714217461057 09091";
    char *msg = "1234";
    char *signature = "26745918808888623786883833832785
    447197125897205360861957116147165712709455207";

    // we verify the signature
    verify_eddsa(out, B1, B2, R1, R2, A1, A2, msg, signature);
}
```

Listing 3: Circuit example for computing 16384 constraints

5 Experiments and Results

In this section, we perform several experiments to prove the efficiency of our solution. In order to do so, we implemented a circuit composed of 16384 constraints as shown in Listing 4.

```
// main function called by the circuit parser
void circuit()
{
    element out;
    // we init the public output
    init_public(&out);

    int mulsize = 16384;
    element arr[mulsize];

    // we init an array of secret elements
    init_array(arr, mulsize);

    // we manually set a value
    input(&arr[0], "12345678");

    // do x multiplications
    for (int i = 1; i < mulsize; i++)
    {
        mul(&arr[i], &arr[1], &arr[i-1]);
    }

    mul(&out, &arr[0], &arr[mulsize-1]);
}
```

Listing 4: Circuit example for computing 16384 constraints

As stated previously, zk-SNARKs are composed of three algorithms. The setup is performed only once, so the performance of this algorithm is not of big importance in practice. However, using `ZPiE`, the setup of a circuit of 16384 constraints can be performed in 17 seconds using a laptop CPU in single-thread. Regarding the proof verification, our zk-SNARK construction has a succinct verifier which verifies any proof in just 0.0011 seconds.

Moreover, we performed several experiments to prove the performance of `ZPiE` when computing proofs. First, we compared our solution with a well-optimized library intended to generate proofs in desktop applications, libsnark. As depicted in Figure 3, `ZPiE` achieves great results, similar to the ones achieved by libsnark, either in single or multi-thread modes.

Then, as depicted in Figure 4, we executed our solution with different constraint amounts in two different processors, either in single and multi-thread modes. As can be seen, the laptop processor i7-11370H (`x86_64`) achieves excellent results either in single or multi-thread modes. Regarding the mobile processor Snapdragon 845 (`aarch64`), the results are still excellent in multi-thread. In single-thread mode, the difference is bigger, yet the proofs can still be executed in a fairly small amount of time.

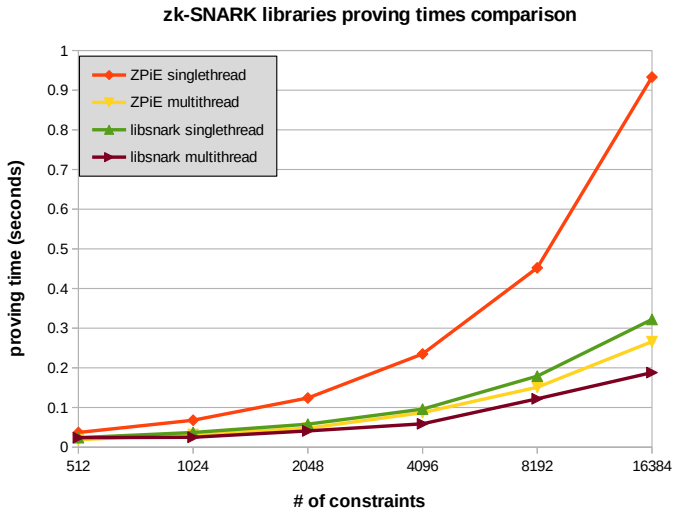

Figure 3: ZPiE and libsnark proving times comparison using different constraint amounts, single-thread and multi-thread modes. All the tests are executed using a i7-11370H CPU. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

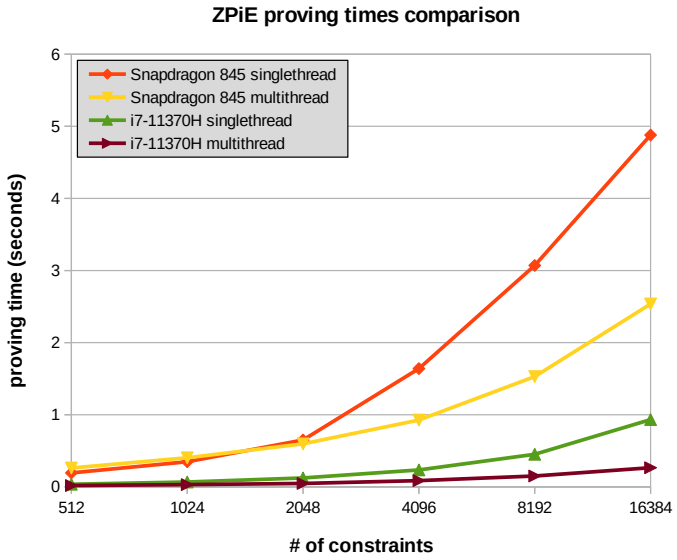

Figure 4: CPUs proving times comparison of ZPiE using different constraint amounts, single-thread and multi-thread modes. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

Furthermore, we successfully computed proofs using a Raspberry Pi Zero W, which CPU architecture is ARM6l (`arm32`) and has a very low clock frequency (700 MHz). As depicted in Figure 5, the results are much higher than using mobile or desktop processors, yet the proofs can still be executed. As such, ZKP applications could be executed in embedded devices, at least when speed is not of paramount importance. For instance, protocols such as `SANS`, which needs around 5000 constraints, could be executed in less than a minute using a Raspberry Pi Zero W. This allows IoT devices to use privacy-preserving protocols based on zk-SNARKs.

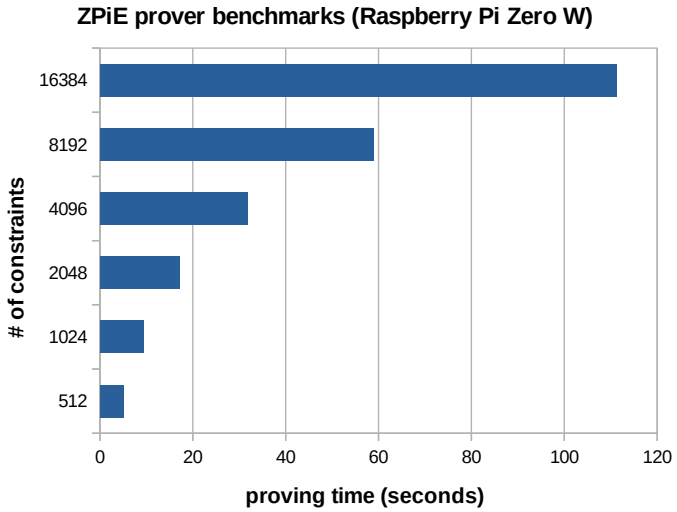

Figure 5: CPUs proving times of ZPiE executed in a Raspberry Pi Zero W. The used zk-SNARK is Groth'16, and the elliptic curve BN128.

6 Conclusions

In this paper, we introduced a novel library intended to generate Zero-Knowledge Proofs in devices with low computing resources. After performing several experiments, the results prove how zk-SNARKs can be computed using processors such as the ARM11 of a Raspberry Pi Zero W. After stating the need for privacy-preserving protocols that use zk-SNARKs in IoT scenarios, we demonstrated that their development and execution in such devices is feasible.

`ZPiE` allows us to develop ZKP applications in embedded devices, by using the specific Groth'16 zk-SNARK. However, we have seen how other schemes exist, each having different features. Implementing other schemes for embedded devices would be an interesting future work to do, in order to expand the uses cases of ZKPs in IoT devices. Overall, being able to use ZKPs in IoT devices should open the door to designing and implementing

more protocols for IoT devices based on ZKPs, not solely for authentication purposes but also for any other privacy matter where ZKPs could be useful.

Moreover, even when we get excellent results, there is still room for improvement. It would be interesting to work on boosting even further the performance of `ZPiE`, by testing other elliptic curve libraries rather than MCL, or by designing new algorithms to reduce the overhead of the executions. Moreover, allowing users to speed up the library using GPUs would be another interesting feature to implement, to allow the usage of our library in scenarios other than IoT devices.

A Groth'16

Let $\mathcal{R}$ be a relation composed of the elliptic curve E over $\mathbb{F}_q$, the pairing e and a QAP $(A, B, C, Z(x))$ representing a circuit. The Groth'16 construction is divided into three algorithms:

- $\sigma \leftarrow Setup(\mathcal{R})$: To perform the setup, we pick $\alpha, \beta, \gamma, \delta, x$ from $\mathbb{F}_r$ and define $\tau = (\alpha, \beta, \gamma, \delta, x)$. Later we compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$:

$$\begin{aligned} vk' &= \{\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\gamma}\}_{i=0}^{l} \\ pk' &= \{\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\delta}\}_{i=l+1}^{m} \\ \sigma_1 &= (\alpha, \beta, \delta, \{x^i\}_{i=0}^{n-1}, vk', pk', \{\frac{x^i t(x)}{\delta}\}_{i=0}^{n-2}) \end{aligned} \tag{3}$$

$$\sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1}) \tag{4}$$

- $\pi \leftarrow Prove(\mathcal{R}, \sigma, s)$ The prover randomly picks r, c in $\mathbb{F}_r$ and computes $\pi = ([\pi_A]_1, [\pi_B]_2, [\pi_C]_1)$:

$$\begin{aligned} \pi_A &= \alpha + \sum_{i=0}^{m} s_i A_i(x) + r\delta \\ \pi_B &= \beta + \sum_{i=0}^{m} s_i B_i(x) + c\delta \\ \pi'_C &= \frac{\sum_{i=l+1}^{m} s_i(\beta A_i(x) + \alpha B_i(x) + C_i(x)) + h(x)t(x)}{\delta} \\ \pi_C &= \pi'_C + \pi_A c + \pi_B r - rc\delta \end{aligned} \tag{5}$$

- $0/1 \leftarrow Verify(\mathcal{R}, \sigma, u, \pi)$: the verifier accepts the proof iff the following equation holds:

$$\begin{aligned} [p_1]_T &= [\pi_A]_1 \cdot [\pi_B]_2 \\ [p_2]_T &= [\alpha]_1 \cdot [\beta]_2 \\ [p_3]_T &= \sum_{i=0}^{l} s_i [\frac{\beta A_i(x) + \alpha B_i(x) + C_i(x)}{\gamma}]_1 \cdot [\gamma]_2 \\ [p_4]_T &= [C]_1 \cdot [\delta]_2 \\ [p_1]_T &= [p_2]_T + [p_3]_T + [p_4]_T \end{aligned} \tag{6}$$

B FFT Techniques for Computing h Coefficients

Our QAP is a 3-matrix set of size $N \times M$. Working on $\mathbb{F}_r$, where r is a prime number and the order of the used elliptic curve, we find a generator g for our field. Having this, we find two values k and an extended size for N, called $N_e = 2^l$ (where l is an integer 'large enough') such that $r = kN_e + 1$. As can be seen, N_e is a power of 2, as 2-addicity is a desirable property for more efficient FFT algorithms. With this, now we can find our N_eth primitive root of unity:

$$\omega \equiv g^k \mod r \tag{7}$$

This generates our domain:

$$S = \{1, \omega, ..., \omega^{N_e - 1}\} \tag{8}$$

Now we need to compute three polynomials $A(z) = \{A_0, ..., A_{M-1}\}, B(z) = \{B_0, ..., B_{M-1}\}, C(z) = \{C_0, ..., C_{M-1}\}$, where z is our toxic waste x defined in the Groth setup:

$$\begin{aligned} A_i(z) &= \sum_{j=0}^{N_e - 1} L_{i,j} \cdot \frac{Lag_j(z)}{z - S_j} \\ B_i(z) &= \sum_{j=0}^{N_e - 1} R_{i,j} \cdot \frac{Lag_j(z)}{z - S(j)} \\ C_i(z) &= \sum_{j=0}^{N_e - 1} O_{i,j} \cdot \frac{Lag_j(z)}{z - S(j)} \end{aligned} \tag{9}$$

where

$$\begin{aligned} Lag_1(z) &= \frac{z^{N_e} - 1}{N_e} \\ Lag_{j+1}(z) &= \omega Lag_j(z) \end{aligned} \tag{10}$$

As such, the setup has defined three polynomials $A(x), B(x), C(x)$. Now, using a solution to our circuit w the prover computes the following values:

$$\begin{aligned} A &= \sum_{i=0}^{M-1} A_i \cdot w_i \\ B &= \sum_{i=0}^{M-1} B_i \cdot w_i \\ C &= \sum_{i=0}^{M-1} C_i \cdot w_i \end{aligned} \tag{11}$$

And we can check:

$$A * B - C = 0 \tag{12}$$

Now, the prover computes the evaluation of three polynomials:

$$\begin{aligned} A_j &= \sum_{i=0}^{M-1} L_{i,j} \cdot w_i \\ B_j &= \sum_{i=0}^{M-1} R_{i,j} \cdot w_i \\ C_j &= \sum_{i=0}^{M-1} O_{i,j} \cdot w_i \end{aligned} \tag{13}$$

Now, the prover will use $A(z), B(z), C(z)$ to compute the coefficients h of $H(x)$:

$$H(x) = \frac{A(x)B(x) - C(x)}{Z(x)} \tag{14}$$

In order to do so, the prover selects a random σ and computes a shifted domain $T = \{\sigma, \sigma\omega, ..., \sigma\omega^{N_e - 1}\}$. He also sets $Z = \sigma^{N_e} - 1$ and does what follows:

- Computes 3 IFFTs in our domain S:

$$\begin{aligned} A_S &= IFFT(A, S) \\ B_S &= IFFT(B, S) \\ C_S &= IFFT(C, S) \end{aligned} \tag{15}$$

- Computes 3 FFTs in our domain T:

$$\begin{aligned} A_T &= FFT(A_S, T) \\ B_T &= FFT(B_S, T) \\ C_T &= FFT(C_S, T) \end{aligned} \tag{16}$$

- Computes $H = \dfrac{A_T B_T - C_T}{Z}$ point by point.
- Computes the shifted coefficients $h_T = IFFT(H, T)$ and it finally gets $h = h_T/\sigma$ point by point.

C Bos-Coster

Let the pairs $(s_1, P_1), (s_2, P_2), ..., (s_n, P_n)$ be the elements of the multi-exponentiations to perform. We sort the list from large to small s_i, by means of a well-optimised binary heap. Then, while the list is larger than 1:

- $(s_1, P_1) = (s_1 - s_2, P_1)$
- $(s_2, P_2) = (s_2, P_1 + P_2)$
- if $s_1 = 0$, we remove this pair
- We sort the list again

When only one pair remains, $s_1 P_1$ is our solution.

Acknowledgements

The authors are supported by Project RTI2018-102112-B-100 (AEI/FEDER, UE).

References

[1] Mansoor Shafi, Andreas F. Molisch, Peter J. Smith, Thomas Haustein, Peiying Zhu, Prasan De Silva, Fredrik Tufvesson, Anass Benjebbour, and Gerhard Wunder. 5g: A tutorial overview of standards, trials, challenges, deployment, and practice. *IEEE Journal on Selected Areas in Communications*, 35(6):1201–1221, 2017.

[2] Ke He, Zizhi Wang, Dong Li, Fusheng Zhu, and Lisheng Fan. Ultra-reliable mu-mimo detector based on deep learning for 5g/b5g-enabled iot. *Physical Communication*, 43:101181, 2020.

[3] Sakshi Painuly, Priya Kohli, Priya Matta, and Sachin Sharma. Advance applications and future challenges of 5g iot. In *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, pages 1381–1384, 2020.

[4] Hamzeh Mohammadnia and Slimane Ben Slimane. Iot-netz: Practical spoofing attack mitigation approach in sdwn network. In *2020 Seventh International Conference on Software Defined Systems (SDS)*, pages 5–13, 2020.

[5] Byoungkoo Kim, Seoungyong Yoon, Yousung Kang, and Dooho Choi. Puf based iot device authentication scheme. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1460–1462, 2019.

[6] Umair Khalid, Muhammad Asim, Thar Baker, Patrick CK Hung, Muhammad Adnan Tariq, and Laura Rafferty. A decentralized lightweight blockchain-based authentication mechanism for iot systems. *Cluster Computing*, pages 1–21, 2020.

[7] Stephan Leible, Steffen Schlager, Moritz Schubotz, and Bela Gipp. A review on blockchain technology and blockchain projects fostering open science. *Frontiers in Blockchain*, 2:16, 2019.

[8] Sovrin Foundation. Sovrin: A Protocol and Token for Self-Sovereign Identity and Decentralized Trust. `https://sovrin.org/wp-content/uploads/Sovrin-Protocol-and-Token-White-Paper.pdf`, Accessed on 28/09/2021, January 2018.

[9] Markus Luecking, Christian Fries, Robin Lamberti, and Wilhelm Stork. Decentralized identity and trust management framework for internet of things. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2020.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[11] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification - Version 2019.0.2, 2019. `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`, Accessed on 28/09/2021.

[12] Xavier Salleras and Vanesa Daza. Sans: Self-sovereign authentication for network slices. *Security and Communication Networks*, 2020, 2020.

[13] D. Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[14] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, Baltimore, MD, August 2018. USENIX Association.

[15] David Gabay, Kemal Akkaya, and Mumin Cebe. Privacy-preserving authentication scheme for connected electric vehicles using blockchain and zero knowledge proofs. *IEEE Transactions on Vehicular Technology*, 69(6):5760–5772, 2020.

[16] Iago Sestrem Ochôa, Valderi Reis Quietinho Leithardt, Leonardo Calbusch, Juan Francisco De Paz Santana, Wemerson Delcio Parreira, Laio Oriel Seman, and Cesar Albenes Zeferino. Performance and security evaluation on a blockchain architecture for license plate recognition systems. *Applied Sciences*, 11(3), 2021.

[17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association.

[18] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2111–2128, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 733–764, Cham, 2019. Springer International Publishing.

[21] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://ia.cr/2019/953`, Accessed on 28/09/2021.

[22] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge snarks for r1cs. Cryptology ePrint Archive, Report 2021/030, 2021. `https://ia.cr/2021/030`, Accessed on 28/09/2021.

[23] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[24] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/2018/046`, Accessed on 28/09/2021.

[25] Eduardo Morais, Tommy Koens, Cees Van Wijk, and Aleksei Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1(8):1–17, 2019.

[26] Shafi Goldwasser and Yael Tauman Kalai. Cryptographic assumptions: A position paper. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography*, pages 505–522, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[27] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[28] 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, pages 103–112, New York, NY, USA, 1988. ACM.

[29] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model. Cryptology ePrint Archive, Report 2017/1050, 2017. `https://eprint.iacr.org/2017/1050`, Accessed on 28/09/2021.

[30] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319–331, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography. Cryptology ePrint Archive, Report 2016/1102, 2016. `https://eprint.iacr.org/2016/1102`, Accessed on 28/09/2021.

[32] Youssef El Housni and Aurore Guillevic. Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security*, pages 259–279, Cham, 2020. Springer International Publishing.

[33] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors, *Pairing-Based Cryptography - Pairing 2010*, pages 21–39, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[35] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2, 2012. `https://cr.yp.to/papers.html#ed25519`, Accessed on 28/09/2021.