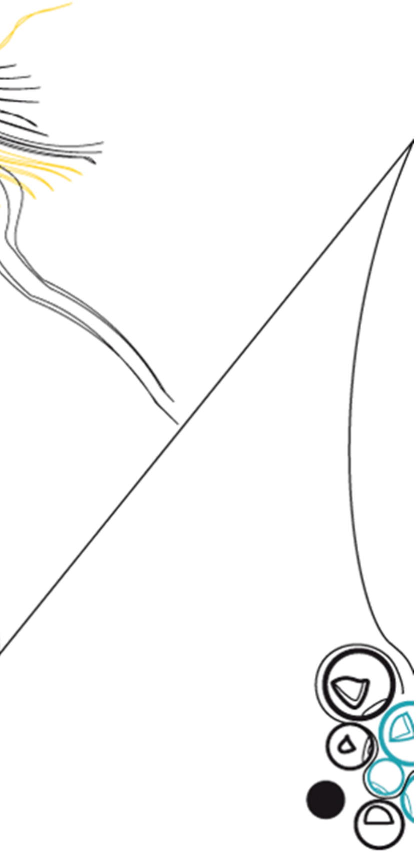
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

zk-SCHNAPS: Enforcing Arbitrary Password Policies in a Zero-Knowledge Password Protocol

Matthijs Roelink

MSc. Thesis
October 2022



Supervisors:
dr. M. H. Everts
prof. dr. ir. R. M. van Rijswijk - Deij

Services and CyberSecurity
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

In this thesis, we introduce zk-SCHNAPS; zero-knowledge Secure Commitment-based Homomorphic Non-interactive Authentication with Passwords using SNARKs. With this password authentication protocol, arbitrary password policies can be enforced by a server, without having the requirement or possibility of inspecting the password. This prevents a server from leaking users' passwords, whether accidentally or on purpose, while still forcing users to choose strong passwords. We do this by using a zk-SNARK to proof compliance of a password during registration, and combining it with a SNARK-friendly encryption scheme (SAVER) to yield an encryption of the password that can be stored by the server. During login, the password is encrypted similar to the SAVER encryption and combined with a zero-knowledge proof, affirming the knowledge of the password that is encrypted. Using the homomorphic property of SAVER, the server can check whether the passwords are equal, without decrypting the individual ciphertexts. We implemented the proposed scheme and show that both proof generation and password verification run in practical time (a few seconds and less than a second respectively) for several real-world password policies, including a blocklist of 100,000 items.

Contents

Abstract **iii**

List of acronyms **vii**

1 Introduction **1**

2 Preliminaries **5**
2.1 Pairings 5
2.2 Homomorphic encryption 5
2.3 Zero-knowledge proofs 6
2.4 zk-SNARKs 6
2.4.1 Pinocchio 7
2.4.2 Overview of zk-SNARK systems 9
2.5 SAVER 11
2.6 Approximate Membership Query Filters 12

3 Related work **15**
3.1 Zero-knowledge password protocols 15
3.2 Password policies 16

4 zk-SCHNAPS **19**
4.1 Main idea 19
4.2 Encoding passwords as input of a zk-SNARK 20
4.3 Encoding password policies in a zk-SNARK 22
4.3.1 Creating password policies 23
4.3.2 Example policies 23
4.4 Protocol 28
4.4.1 Setup 29
4.4.2 Registration 29
4.4.3 Login 31
4.4.4 Change password 33
4.5 Protecting against replay attacks 36

4.6 Security 38

5 Evaluation 41

5.1 Implementation 41

5.2 Performance 42

5.2.1 schnapsjs 42

5.2.2 Password blocklist 44

6 Discussion and Future Work 49

6.1 Password hash function 49

6.2 Proving speed 50

6.3 Fetching salts 50

7 Conclusion 51

References 53

List of acronyms

AMQ-Filter	Approximate Membership Query Filter
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BPR	Blind Password Registration
CRS	Common Reference String
EKE	Encrypted Key Exchange
HMAC	Hash-based Message Authentication Code
MFA	Multi-Factor Authentication
NIST	National Institute of Standards and Technology
PAKE	Password-Authenticated Key Exchange
SAVER	SNARK-friendly, Additively-Homomorphic and Verifiable Encryption and decryption with Rerandomization
SRP	Secure Remote Password
UTF-8	UCS Transformation Format 8
ZKPPC	Zero-Knowledge Password Policy Checks
zk-SCHNAPS	zero-knowledge Secure Commitment-based Homomorphic Non-interactive Authentication with Passwords using SNARKs
zk-SNARK	zero-knowledge Succint Non-interactive ARgument of Knowledge

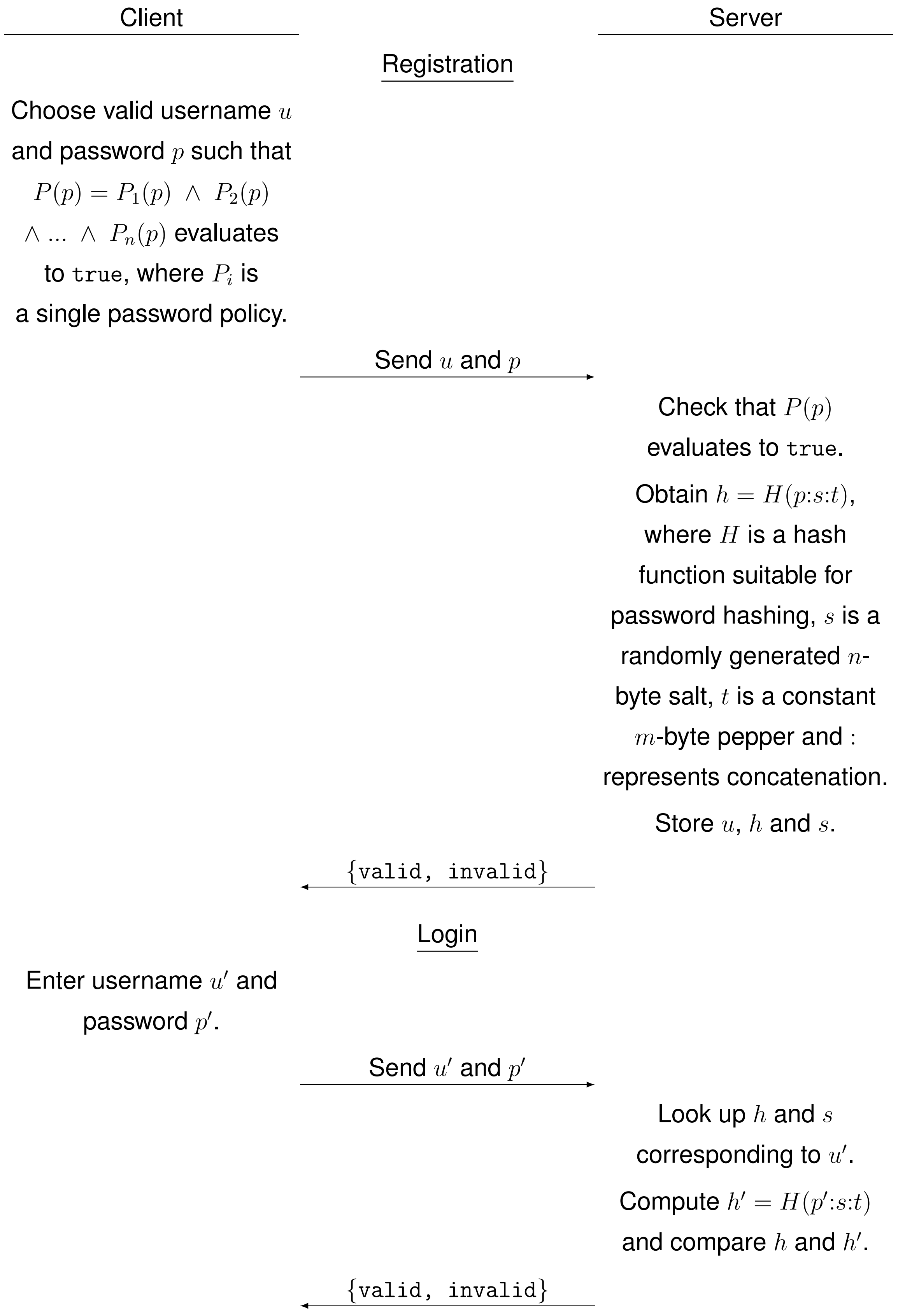
Chapter 1

Introduction

Passwords are still the dominant means of authenticating a user to a (web) service. During registration, a user chooses a username and password of their liking and sends their credentials to the service. If the service approves the credentials (e.g. the username is not yet taken, the password has at least eight characters), the password is hashed and stored alongside the username, often in a database. Hashing is required to prevent disclosure of plaintext passwords due to a data leak. Before hashing, a randomly generated salt is added, such that same passwords do not result in the same hash. This salt is then stored alongside the password. In addition, a constant pepper is added, which is exclusively known to the server and prevents bruteforcing passwords if only the database is leaked. After the registration phase, the chosen credentials can be used to subsequently log in to that service by sending them in plaintext. This typical registration and login workflow is also shown in Figure 1.1.

A service approves a password during registration if it complies with certain requirements, also known as *password policies*. Examples of such policies are that the password needs to have a certain minimum length or a certain minimum number of characters from a specific group, such as alphanumeric and special characters. In addition, commonly used passwords may be rejected by using a blocklist, on which the chosen password may not occur. These policies are intended to force users to choose more difficult-to-guess passwords, which prevents unauthorized parties to log in using a guessed password.

For a long time, passwords were assumed to be secure enough. For example, access to online bank accounts was granted using only a username and password, without additional security measures. However, due to carelessness of users, passwords are not always chosen securely and are shared among multiple services. In addition, due to data breaches, passwords have become available to adversaries in large numbers. Thus, adversaries logging in using someone else's credentials is not unimaginable.



Client

Server

Registration

Choose valid username u and password p such that $P(p) = P_1(p) \wedge P_2(p) \wedge ... \wedge P_n(p)$ evaluates to `true`, where P_i is a single password policy.

Send u and p

Check that $P(p)$ evaluates to `true`.

Obtain $h = H(p{:}s{:}t)$, where H is a hash function suitable for password hashing, s is a randomly generated n-byte salt, t is a constant m-byte pepper and : represents concatenation.

Store u, h and s.

{`valid, invalid`}

Login

Enter username u' and password p'.

Send u' and p'

Look up h and s corresponding to u'.

Compute $h' = H(p'{:}s{:}t)$ and compare h and h'.

{`valid, invalid`}

Figure 1.1: Classic registration and login workflow with passwords

In recent years, a shift towards multi-factor authentication (MFA) has strengthened service authentication by requiring another factor of authentication. While passwords are considered to be something you know, with MFA it needs to be complemented by something you have (typically a one-time password on a trusted device) or something you are (using biometrics). Only knowing the password is thus not sufficient anymore to log in, making it more difficult for adversaries to break into someone else's account.

While MFA enhances security, it does not tackle some of the issues of passwords. First of all, a user has to trust the service with their password. Passwords are generally sent in plaintext to the service. This means that the service is able to inspect the password, which is needed to determine whether the password meets the password policy requirements in the registration phase, or whether the password is indeed the correct password in the login phase. However, since the service knows the plaintext password, it could share it with third parties or sell it to adversaries without the user knowing.

Furthermore, the service is trusted to securely store the password. It is common practise to apply a memory-hard hash function (such as *Argon2* [BDK16] or *scrypt* [Per09]) together with a salt and pepper to the password, such that the resulting hash is of limited use to an attacker, while still being able to validate a password in the login phase. However, it has been shown multiple times that some services store their users' passwords in plaintext (e.g. [Ng19]) or use an insecure hash function, risking password leaks.

To counter these issues, zero-knowledge password protocols were created. In a zero-knowledge password protocol, a service can be convinced that the user knows their password, without needing to send it. This solves the two problems discussed above, since the password is never sent to the server. However, zero-knowledge password protocols bring their own challenges. Because the password is not sent to the server, it is not straightforward to enforce password policy requirements anymore. In addition, most protocols require interactivity, hence increasing the communication complexity.

In 2014, [KM14] introduced a new class of protocols, namely Zero-Knowledge Password Policy Checks (ZKPPC), which allows for basic password policy enforcement while still preserving zero-knowledge of the password. This allows the server to be convinced of the password strength without inspecting the password. However, [KM14] only supports a limited set of password policies. These policies are minimum password length, maximum password length and requirements regarding the number of uppercase and lowercase letters, digits and symbols in a password. Other policies, such as prohibiting certain passwords (e.g. 'password123'), are not supported, while they can greatly increase the security.

To fill this gap, we propose zk-SCHNAPS, zero-knowledge Secure Commitment-based Homomorphic Authentication with Passwords using SNARKs, which is a non-interactive zero-knowledge password protocol that supports arbitrary password policies. We do this by leveraging zk-SNARKs to prove compliance to the implemented password policies. We provide an implementation of the protocol and show that it runs in practical runtime for several implemented password policies.

This thesis is structured as follows. Chapter 2 provides some preliminaries. Then, Chapter 3 describes the related work. Chapter 4 proceeds by describing the zk-SCHNAPS protocol. Chapter 5 evaluates the protocol by providing an implementation and showing benchmarks. Chapter 6 then discusses the results and provides directions for future work. Finally, Chapter 7 concludes the paper.

Chapter 2

Preliminaries

2.1 Pairings

Definition 1. *Let $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$ be cyclic groups of prime order q. A pairing is a map*

$$e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$$

that satisfies the following properties:

1. ***Bilinearity**: For all $u \in \mathbb{G}_0$, $v \in \mathbb{G}_1$ and $a, b \in \mathbb{Z}$ we have that*

$$e(u^a, v^b) = e(u, v)^{ab}$$

2. ***Non-degeneracy**: For generators $g \in \mathbb{G}_0$ and $h \in \mathbb{G}_1$, we have that $e(g, h) \neq 1$.*

2.2 Homomorphic encryption

We use the following definition of homomorphic encryption:

Definition 2. *A homormophic encryption scheme is an encryption scheme with operations $\otimes$ and $\oplus$ such that*

$$E(m_1) \otimes E(m_2) = E(m_1 \oplus m_2)$$

for all plaintexts m_1 and m_2. An encryption scheme is said to be additive homomorphic if there is an operation $\otimes$ such that

$$E(m_1) \otimes E(m_2) = E(m_1 + m_2)$$

for all plaintexts m_1 and m_2. Similarly, an encryption scheme is multiplicative homomorphic if there is an operation $\otimes$ such that

$$E(m_1) \otimes E(m_2) = E(m_1 \cdot m_2)$$

for all plaintexts m_1 and m_2.

Homomorphic encryption has several applications. One application is that of private data aggregation in smart grids, where homomorphic encryption is used to calculate the sum of a neighborhood's or single household's consumption, without having access to the individual readings [ET12, Erk15]. This allows the provider to invoice households based on their exact consumption in a privacy-friendly manner.

Another application is that of multi-party computation, where different parties together compute a function over their inputs, without exposing those inputs to the other parties. Homomorphic encryption enables this by encrypting the inputs and performing calculations in the encrypted domain, after which the result can be decrypted to obtain the output [AW21].

2.3 Zero-knowledge proofs

Zero-knowledge proofs were already designed in 1989 [GMR89] and are a type of proof in which a prover can convince a verifier that a given statement is true without revealing any additional information about the statement. Zero-knowledge proofs can thus be effective in situations where a prover wants to convince a verifier it knows a certain secret value, without exposing it to the verifier.

In order for a proof to be zero-knowledge, it needs to comply to three requirements: *completeness*, *soundness* and *zero-knowledge*. These requirements ensure that the proof is secure and does not leak any additional information besides the proven statement. *Completeness* refers to the fact that if the prover knows the secret, then the verifier accepts their proof with probability one. Hence, if the prover creates a valid proof, the verifier should always be able to successfully verify the proof. The second requirement is *soundness* and is concerned with the small probability that the prover does not know the secret, but is still able to create a valid proof and thus convince the verifier. Naturally, this probability should be very small to provide a secure zero-knowledge scheme. Finally, the *zero-knowledge* requirement refers to the fact that nothing can be learnt from the proof, except that the proven statement is true. This implies that the proof does not leak any information about the secret of the prover. This property is often proven by comparing simulations (in which the secret is not known) and valid transcripts, and proving that these are indistinguishable from each other.

2.4 zk-SNARKs

First described in 2012 by [BCCT12], zk-SNARKs are a class of zero-knowledge proofs that have certain properties, as dictated by its acronym: **z**ero-**k**nowledge

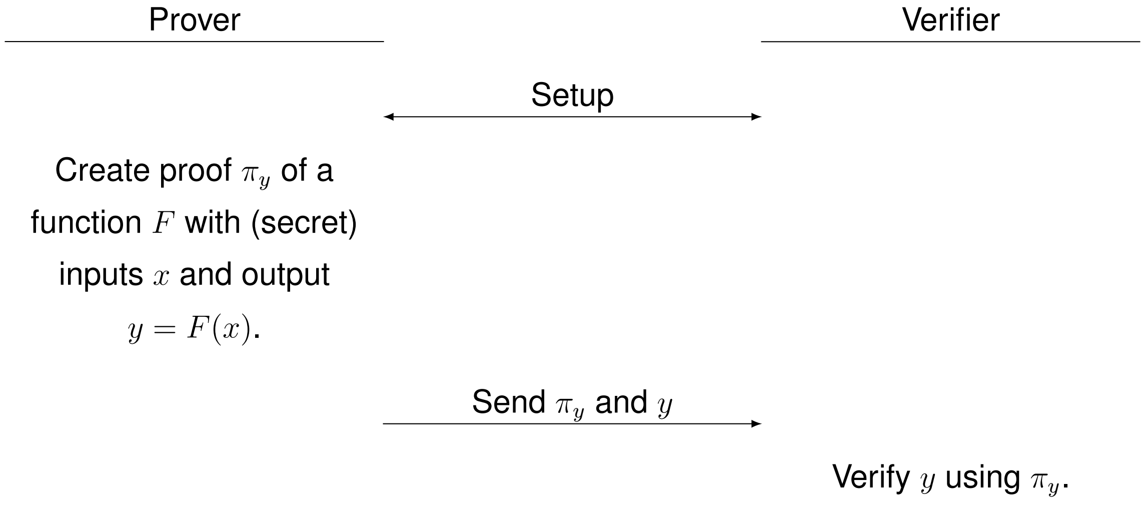

Prover
Verifier
Setup
Create proof π_y of a function F with (secret) inputs x and output $y = F(x)$.
Send π_y and y
Verify y using π_y.

Figure 2.1: zk-SNARK stages

Succinct **N**on-interactive **AR**gument of **K**nowledge. *Zero-knowledge* refers to the fact that given a zk-SNARK proof, no additional information can be learnt about the secret than the statement that is proven. *Succinct* refers to the proof size and verification time of the proof. In order for a proof to be succinct, it needs to be small (only a few hundred bytes for large programs) and have a short verification time in the order of milliseconds. The *non-interactive* property indicates that there is no interaction required between the prover and the verifier, which means that the prover can create a proof without involving the verifier. Finally, the *argument of knowledge* part indicates that the prover is able to convince the verifier that they know a secret, without revealing that secret.

In general, a zk-SNARK proving system consists of three stages, namely `setup`, `prove` and `verify`, as shown in Figure 2.1. In the `setup` phase, the proof parameters are constructed. Depending on the specific scheme, this phase is required per function, one-time only or not at all. In addition, the setup phase may be executed solely by the verifier and shared with the prover, or in collaboration with the prover using for example multi-party computation. In the `prove` phase, the prover uses the parameters generated in the `setup` phase to create a proof π_y of a function F with (secret) inputs x and output $y = F(x)$. Together with the output, the proof is sent to the verifier, who initiates the `verify` phase. In this phase, the proof π_y is verified using the setup parameters and the result y, resulting in `valid` or `invalid`.

2.4.1 Pinocchio

The *Pinocchio* system presented in [PHGR13] is the first zk-SNARK with nearly practical performance and forms the basis of many current zk-SNARK systems. In

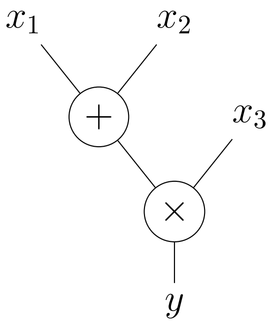


Figure 2.2: Example of an arithmetic circuit with inputs x_1, x_2 and x_3 and output y. This arithmetic circuit computes $y = (x_1 + x_2) * x_3$.

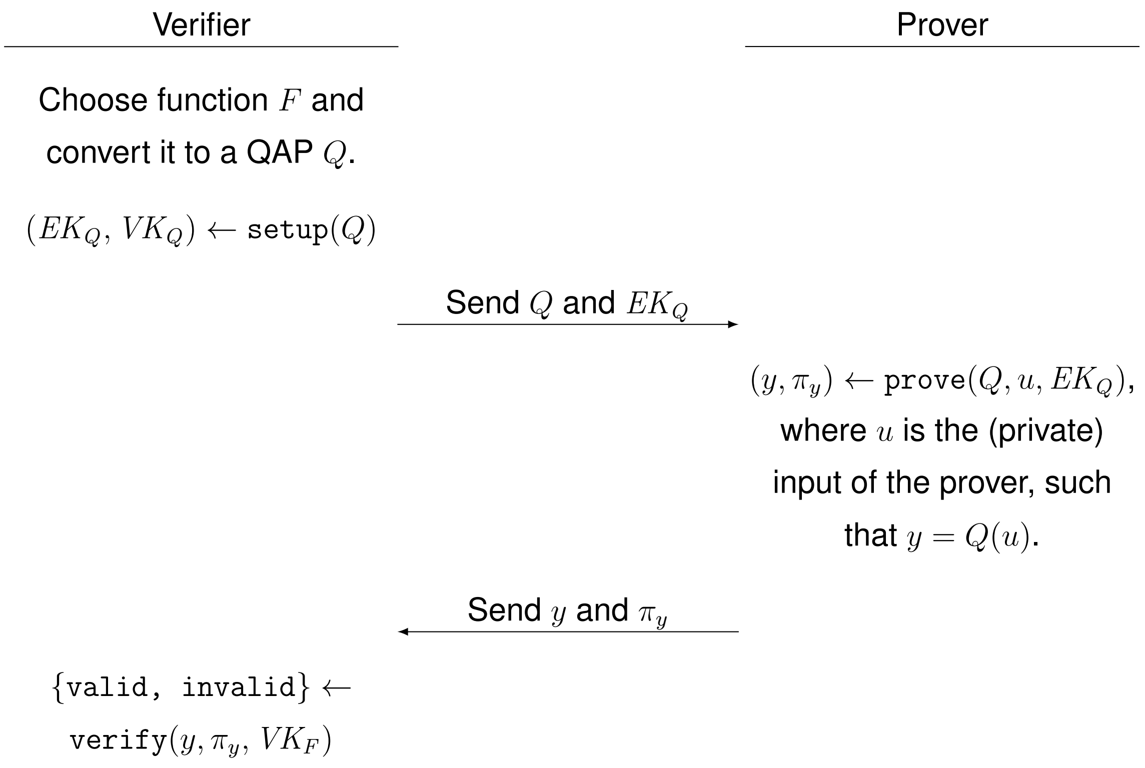
their paper, the authors present a solution to achieve verifiable computation in a more efficient way. The goal of verifiable computation is to provide a proof of a certain computation, which, together with the output of the computation, can be checked for validity by a different party. The typical setting for verifiable computation is outsourced computations from weaker clients (such as smartphones) to more powerful workers (such as cloud servers). In such a setting, the client needs a guarantee that the result returned from the worker is indeed the outcome of the computation, which it can get by verifying the proof.

In order to prove the output of a function F, F needs to be converted to a Quadratic Arithmetic Program (QAP), which is a way to encode an arithmetic circuit. An arithmetic circuit is a circuit with input and output wires that connect to addition or multiplication gates, where values flow from the inputs to the outputs. An example of an arithmetic circuit is shown in Figure 2.2. Since [GGPR13] shows how an arithmetic circuit representing F can be efficiently encoded into a QAP, it suffices to convert F to an arithmetic circuit. Hence, the workflow to encode F into a QAP is as follows:

$$F \xrightarrow{convert} \text{arithmetic circuit} \xrightarrow{convert} \text{QAP}$$

Before the prover can create a proof for their computation, a trusted setup is required to generate the public evaluation key EK_Q and the public verification key VK_Q for a QAP Q. Note that Q is required in the setup phase, and hence Pinocchio requires a per-program setup. The setup can be done by the verifier, or in collaboration with the prover. The prover cannot perform the setup on their own, because generating the evaluation and verification keys exposes the randomly sampled values, which are also known as *toxic waste*. With this toxic waste, the prover is able to forge proofs, which eliminates the trust in the proofs.

When the keys have been generated in the `setup` phase, Q and EK_Q are sent to the prover, as shown in Figure 2.3. The prover uses its own input u to compute



Verifier

Prover

Choose function F and convert it to a QAP Q.

$(EK_Q, VK_Q) \leftarrow \texttt{setup}(Q)$

Send Q and EK_Q

$(y, \pi_y) \leftarrow \texttt{prove}(Q, u, EK_Q)$, where u is the (private) input of the prover, such that $y = Q(u)$.

Send y and π_y

$\{\texttt{valid, invalid}\} \leftarrow \texttt{verify}(y, \pi_y, VK_F)$

Figure 2.3: The workflow of Pinocchio

$y = Q(u)$ and a proof π_y, which is an attestation that y is indeed the outcome of the computation. The size of π_y is always 288 bytes and thus independent of the size of the computation. After y and π_y have been computed, they are sent back to the verifier. Using VK_F and π_y, it can be verified whether y was indeed the outcome of the computation.

2.4.2 Overview of zk-SNARK systems

Since the publication of [PHGR13], the field of zk-SNARKs has seen an increase in research activity with many zk-SNARK systems that improve upon Pinocchio on several aspects, such as the proving efficiency and improvements on the `setup` phase. In general, we can make a distinction between circuit-dependent and universal zk-SNARKs.

2.4.2.1 Circuit-dependent zk-SNARKs

These zk-SNARK systems are characterized by the fact that a trusted setup is required for each circuit. While this limits the usability of these systems, the proving time and size of the proofs are typically smaller than those of universal systems. The Pinocchio scheme discussed in Section 2.4.1 falls under this category.

In 2015, [CFH$^+$15] published their paper about *Geppetto*, which improves on Pinocchio by introducing Multi-QAPs. Multi-QAPs enable the prover to commit to the same data in multiple proofs. This is especially useful in loops, in which the loop body can be committed to once instead of unrolling the loop.

The most efficient zk-SNARK to date is that described by [Gro16]. While Pinocchio's proofs consist of 8 group elements, [Gro16] manages to reduce this to only 3 group elements, effectively reducing the proof size. In addition, verification of proofs is faster with only 3 pairings, as opposed to Pinocchio's 11 pairings. Finally, the keys generated in the `setup` phase are smaller.

Pinocchio has already seen practical use in the digital currencies Zerocash [BSCG$^+$14] and its successor ZCash [Zer19], where zk-SNARKs are used to provide strong privacy guarantees. Because of its efficiency, [Gro16] has replaced the Pinocchio proving system in the current version of ZCash.

2.4.2.2 Universal zk-SNARKs

Universal zk-SNARKs do not require a trusted setup per circuit and are thus universal in use. Zk-SNARKs in this category can be subdivided into three categories, namely *universal setup*, *transparent setup* and *universal circuits*.

Universal setup. Zk-SNARKs with univeral setup are characterized by requiring one universal trusted setup for all circuits. *Sonic*, introduced in [MKBM19], has a universal setup, enabling proving any program up to a certain size. In addition, its setup can be updated afterwards, strengthening its security in case the toxic waste of the setup is leaked. While having competitive cost of verification, Sonic requires batching of proofs to be efficient. Other zk-SNARK system in this category are *Libra* [XZZ$^+$19], *PLONK* [GWC20], *Marlin* [CHM$^+$20], *Mirage* [KPPS20] and *Lunar* [CFF$^+$21].

Transparent setup. Zk-SNARK systems in this category do not require a trusted setup, but at most a transparent setup. A transparent setup does not produce any toxic waste, and can thus be publicly performed. However, the proof size is typically larger than systems with a trusted setup, in the order of tens or hundreds of kilobytes. One system with a transparant setup is *Aurora* [BSCR$^+$19], in which only the hash function needs to be agreed on by the parties, which is public information. Other zk-SNARK systems with a transparent setup are *Ligero* [AHIV17], *Spartan* [Set20], *Halo* [BGH19], *Fractal* [COS20], *Hyrax* [WTS$^+$18] and *zk-STARK* [BSBHR18].

Universal circuits. The last category of universal zk-SNARKs are that of universal circuits. Instead of converting each to-be-proven program to an arithmetic circuit, this category of systems aims to generate one universal circuit for all programs. This is done by viewing a program as data fed into the universal circuit, effectively simulating a processor. [BSCTV14] is such a system, in which programs are executed on *vnTinyRAM*, a von Neumann RISC architecture. While any program up to a certain size can be proven with this method, the system has a clock rate of verified instruction per second of around 0.1Hz, making it unsuitable for practical use in its current form.

2.5 SAVER

Presented in [LCKO19] in 2019, SAVER (*SNARK-friendly, Additively-homomorphic and Verifiable Encryption and decryption with Rerandomization*) provides a solution to combine encryption with zk-SNARKs. Traditionally, in order to encrypt inputs or outputs of a zk-SNARK, the encryption should be performed in the zk-SNARK circuit. However, due to the complex cryptographic operations, this increases the circuit size in such an extent that the proving time becomes impractical. Instead, SAVER tackles this problem by encrypting outside the circuit, and linking the encryption to the zk-SNARK. This does not add any cryptographic computations to the zk-SNARK circuit, and hence does not impact proving time.

SAVER is a probabilistic encryption scheme and can be used in conjunction with [Gro16], as well as with other pairing-based schemes such as [GM17] and [KLO20]. As its expanded name suggests, it provides verifiable encryption and decryption, and also supports rerandomization. Next to that, SAVER is additively homomorphic, meaning that two encryptions $c_1 = E(m_1)$ and $c_2 = E(m_2)$ can be combined into $c_{12} = E(m_1 + m_2)$ by multiplying c_1 with c_2.

In contrast to most encryption schemes, SAVER's decryption does not yield the original plaintext m, but g^m, where g is an element resulting from a pairing. Thus, after the actual decryption, the result should be bruteforced to find the original plaintext m. To make this efficient, the original plaintext should be split into n blocks, such that each plaintext part m_i can efficiently be bruteforced. The parts $m_1, \ldots, m_n$ can then be combined together to form the original plaintext.

The following algorithms are provided by SAVER:

- SETUP($relation$): performs the zk-SNARK setup and prepares the Common Reference String (CRS).
- KEYGEN(CRS): generates three SAVER keys, namely a public key, secret key and verification key.

- ENC($CRS, PK, m_1, \ldots, m_n, \phi_{n+1}, \ldots, \phi_l; w$): encrypts the inputs $m_1, \ldots, m_n$ and returns the ciphertext together with the zk-SNARK proof.

- RERANDOMIZE($PK, \pi, \mathcal{CT}$): rerandomizes the ciphertext.

- VERIFY_ENC($CRS, PK, \pi, \mathcal{CT}, \phi_{n+1}, \ldots, \phi_l$): verifies the validity of the ciphertext and the proof.

- DEC($CRS, SK, VK, \mathcal{CT}$): decrypts the ciphertext.

- VERIFY_DEC($CRS, VK, m_1, \ldots, m_n, \nu, \mathcal{CT}$): verifies the validity of the decrypted ciphertext.

2.6 Approximate Membership Query Filters

An Approximate Membership Query Filter (AMQ-Filter) is a data structure that stores data in a space-efficient way, such that it can be efficiently determined whether an element is present in the filter. Since an AMQ-Filter is a probabilistic data structure, it is possible that an element not present in the filter is designated as in the filter with probability ϵ. While false positives can thus occur, false negatives are impossible. Hence, if the answer to a membership query returns `not in filter`, it is certain that the element is indeed not in the filter. In contrast, when returning `in filter`, it is with probability $1 - \epsilon$ that this is indeed the case.

There are several types of AMQ-Filters with their own strengths and weaknesses. The most notable are the Bloom filter [Blo70], cuckoo filter [FAKM14] and xor filter [GL20].

Already conceived in 1970 [Blo70], a Bloom filter consists of a sequence of m bits, all set to 0 initially. These bits are mutated using k different hash functions, each mapping an arbitrary input to an index in the sequence. When an element is inserted into the Bloom filter, it is hashed by each of the k hash functions, resulting in k indices. The bit at each of those indices is then set to 1, regardless whether it is already set to 1. Looking up an element then consists of hashing the element with the k hash functions and checking whether all bits at the corresponding indices are set to 1. If this is the case, the element is probably in the filter. If not, the element is definitely not in the filter.

The cuckoo filter was published in 2014 [FAKM14] and presented as a replacement for Bloom filters. While Bloom filters only support insertion and lookup, cuckoo filters also support deletion. Instead of single bits, cuckoo filters store fingerprints of the inserted elements in n buckets with bucket size b. It uses exactly two hash functions and cuckoo hashing [PR01] to insert these fingerprints in the filter. When inserting an element, it is hashed with the first hash function, resulting in an index referring to one of the n buckets. If the bucket is not yet full, its fingerprint is computed and inserted into the bucket. Otherwise, the element is hashed with the second function and its fingerprint is inserted into the designated bucket if not full. If both buckets are full, one of the fingerprints in one of the two buckets is replaced with the new fingerprint. The alternative location of the replaced fingerprint is calculated and the above procedure is repeated with the replaced fingerprint, until either all fingerprints are situated in a bucket, or it is concluded that the insertion failed after reaching a predefined maximum iterations. Looking up an element consists of locating the two possible buckets and checking whether one of the buckets contains the fingerprint of the element. Removing an element from the cuckoo filter is achieved by removing the fingerprint of that element from one of the two candidate buckets.

In contrast to the previously discussed filters, an xor filter [GL20] only supports inserting elements during its construction. Similar to the cuckoo filter, it stores fingerprints of elements, but these are stored in a one-dimensional array instead of buckets. During construction, three hash functions are chosen, each mapping an element to a location in one-third of the array of fingerprints. By cleverly constructing the xor filter, element lookup can be performed by a series of hash and XOR operations. During lookup, the fingerprint of the element is computed and the element is hashed using the three hash functions. Each hash refers to a location in the array of fingerprints, and the corresponding fingerprints are XOR'ed. If the result of these XOR operations is equal to the fingerprint of the element, the element is probably in the filter. Otherwise, it is definitely not in the filter.

While the aforementioned AMQ-Filters are all achieving a common goal, their features and performances differentiate them. It has been shown that Bloom filters require around $1.44 \cdot \log_2(\frac{1}{\epsilon})$ bits per inserted element [Blo70], where ϵ is the false positive rate. In contrast, cuckoo filters require less bits per element for a small ϵ, namely $\frac{\log_2(\frac{1}{\epsilon})+2}{\alpha}$ [FAKM14], where α is the load factor of the filter. In addition, cuckoo filters require only two hash functions, as opposed to k hash functions of a Bloom filter. Finally, xor filters are shown to require $1.23k$ bits per element, or $1.0824k + 0.5125$ in its optimized form, where k is the fingerprint size in bits, beating the Bloom filter and cuckoo filter when requiring a small false positive rate. Similar to the cuckoo filter, only a small number of hash functions are required, namely three.

Chapter 3

Related work

3.1 Zero-knowledge password protocols

The National Institute of Standards and Technology (NIST) describes a zero-knowledge password protocol as “[a] password-based authentication protocol that allows a claimant to authenticate to a verifier without revealing the password to the verifier” [GGF17]. This means that, while passwords are used in the protocol, the verifier never learns the password, but can be convinced of its correctness.

The basis of zero-knowledge for authentication has been laid already in 1992 with the publication of [BM92]. In their paper, the authors introduce the *Encrypted Key Exchange* (EKE), which is a secure interactive key exchange using a password as encryption key. Since messages are encrypted, parties without knowledge of the password are not able to authenticate. The shared key originating from the key exchange can subsequently be used to create zero-knowledge proofs, as knowledge of the key proofs knowledge of the password. One downside, however, is that both parties in the authentication process need to know the password in order to perform the key exchange.

An important zero-knowledge password protocol still used today is the *Secure Remote Password* (SRP) introduced in [Wu98]. In SRP, there are two parties, namely a client that knows the password, and a server that holds a *verifier*. The verifier is computed by the client and sent to sever, with which it can authenticate the client. The verifier cannot be used to find the password or impersonate as the client, making it useless if the verifier is compromised. Since the server does not learn the password, SRP is a zero-knowledge protocol.

Both protocols described above can be generalized as *Password-Authenticated Key Exchange* (PAKE) protocols, in which a password is used to securely establish a shared key interactively. There are two types of PAKE protocols, namely *balanced PAKE* and *augmented PAKE*. Balanced PAKE assumes that both parties have knowledge of the password. EKE is thus a balanced PAKE. Other balanced PAKE protocols are *SPEKE* [Jab96], *J-PAKE* [HR10] and *CPace* [AHH22]. Augmented PAKE on the other hand only requires one party to know the password, making this type more suitable for use in zero-knowledge protocols. SRP falls under this type, as well as *Augmented EKE* [BM93], *AuCPace* [HL19] and *OPAQUE* [JKX18]. CPace and OPAQUE have been standardized by the IETF in 2020 [Smy20].

Because the server does not receive the password in augmented PAKE protocols, it cannot enforce password policies such as minimum password length. [KM14] introduces *Zero-Knowledge Password Policy Checks* (ZKPPC), which is a class of protocols that preserve zero-knowledge of the password, while still being able to perform basic policy checks. The supported password policies are limited to minimum and maximum length and requirements regarding the number of uppercase and lowercase letters, digits and symbols in a password. In addition, only ASCII characters are supported.

In a later paper, the same authors introduced *Blind Password Registration* (BPR) [KM16], improving upon [KM14]. While BPR is simpler and faster, it still only supports the same limited password policies, except that the constraint on the maximum password length is removed. To the best of this author's knowledge, no other work about enforcing password policies in PAKE protocols exists.

3.2 Password policies

Several studies, such as [FH07] and [DMR10], have shown that people tend to use passwords that are easy to remember, ranging from common passwords, such as 'password', to using their name or other public information as their password. However, those passwords are also much easier to guess by adversaries, which enables them to impersonate other users.

To counter this, it is common to create rules that a password should comply with, so-called *password policies*. These password policies are enforced by a service during registration, such that users can only register if they choose a strong password. The aim of password policies is to force users to choose more complex passwords, typically with a certain minimum length and different character sets. [FH10] defines three kind of attacks that are reduced by enforcing password policies: online brute-force attacks, offline attacks on the file of hashed passwords and password re-use across sites.

In online brute-force attacks, an adversary tries to guess the password by trying many passwords for a single account. This can be done by iterating over all possible passwords (e.g. 'aaa', 'aab', ...), as well as using a dictionary of common passwords. This last attack is also known as a *dictionary attack*. While most systems lock an account after a certain number of incorrect password guesses, an adversary might still be able to guess credentials of some account by fixing a handful of passwords and iterating over many usernames. If accounts have a weak password, the adversary might thus be able to learn those credentials, provided that they can obtain a large number of usernames. Password policies can prevent this type of attack by eliminating weak passwords, hence making bruteforcing unfeasible.

Secondly, with offline attacks on the file of hashed passwords it is assumed that an adversary has gained access to the collection of hashed passwords and tries to find the passwords corresponding to the hashes. While there are ways to make it harder for an adversary to obtain a password from its hash using bruteforce (such as using a salt and pepper in the hash and using memory-hard hash functions), it is still relatively easy for an adversary to find weak passwords. These weak passwords can be prevented by using strict password policies.

Finally, password re-use can be reduced because of different policies. The threat of password re-use is that when the password is leaked from an account from one service, other accounts of that user with the same password are also vulnerable. By having different password policies, it might be that certain passwords are allowed at one service, but not at another. For example, one service might impose a constraint that no digits are allowed in the password, while another service might require at least two digits. However, as the authors also notice, this does not seem to be a primary goal of password policies.

To measure the effect of password policies, [KSK$^+$11] uses the definition of *entropy* to measure the password strength resulting from several policies. The entropy of a password is correlated with the amount of guesses that an adversary needs to do at most in order to find the password. If a password has an entropy of H, an adversary needs to make 2^H guesses at most, or 2^{H-1} guesses on average. The entropy of a password can thus be calculated using the following formula:

$$H = \log_2 (N^L),$$

where N is the number of possible characters (e.g. digits and/or letters) and L the password length. The higher the entropy, the more guesses an adversary needs on average and thus the stronger the password is.

In their study, [KSK$^+$11] argues that password policies should be a balance between security, where strict password policies improve password entropy and thus password strength, and usability, where strict password policies degrade the usability because users have to come up with and remember more difficult passwords. In addition, the authors show that dictionary checks (prohibiting dictionary words) increase the entropy of a password, but less than expected, while significantly decreasing the usability. From their tested password policies, it is concluded that a minimum password length of 16 characters without additional rules provides the most entropy, while preserving usability.

Another research on password policies is [SKD+16], which aims to design secure password policies, while also limiting its impact on usability. Through a large-scale study, the password strengths originating from several policies are examined. In contrast to [KSK+11], the strength of a password is not measured using entropy, but using the number of required guesses to crack a password hash using two algorithms. Next to insights in the nature of chosen passwords by the participants, the authors describe some recommendation for service providers, among which avoiding relying exclusively on length-only requirements and using substring blocklists (e.g. disallowing passwords that contain the sequence ‘123’).

Chapter 4

zk-SCHNAPS

In this section, we present zk-SCHNAPS, zero-knowledge Secure Commitment-based Homomorphic Authentication with Passwords using SNARKs. With this password authentication scheme, users can authenticate to a server without exposing their password to the server. In addition, arbitrary password policies can be enforced in zero-knowledge. This chapter is structured as follows. Section 4.1 presents the main idea of the protocol. Section 4.2 then proceeds by discussing the encoding of passwords in the zk-SNARK. Section 4.3 explains how password policies can be designed and provides examples of the most common password policies. Section 4.4 then explains the proposed protocol in detail. Section 4.5 proceeds by discussing measures against replay attacks. Finally, Section 4.6 discusses the security of the used components and the protocol.

4.1 Main idea

The zk-SCHNAPS protocol consists of three phases, namely *registration*, *login* and *change password*. Before the protocol can be used, a *setup* should be executed by the server, establishing the necessary cryptographic primitives.

In the registration phase, a user registers theirself by choosing a username and password. In order to be accepted by the system, the password should comply to predefined password policies to ensure a secure password. These password policies are embedded in a zk-SNARK, which proves compliance without exposing the password. Because of its efficiency, the zk-SNARK system described in [Gro16] is used, which has a proof size of only a few hundred bytes. The zk-SNARK is combined with SAVER to encrypt the password hash for storage. The registration phase is only executed once per user; after registration, users can advance to the login phase.

During the login phase, a user enters their username and password as chosen in the registration phase. Because the password was already confirmed to comply to the password policies in the registration phase, such a check is not necessary anymore, and hence no zk-SNARK is used in this phase. Instead, the password is encrypted with SAVER and combined with a zero-knowledge proof, affirming knowledge of the password that is encrypted. Using the homomorphic property of SAVER, the two encryptions are combined by the server using division, after which it can check whether the passwords are equal by decrypting the combined ciphertext and checking whether the result is equal to 1. This works because dividing two encryptions of the same password hash results in the encryption of 0, which, due to SAVER's decryption resulting in g^m for a base g and plaintext m, decrypts to 1. Only if the username and password matches those chosen in the registration phase, the decryption result will be 1 and the user can successfully login.

When a user is logged in, they can optionally change their password. This phase is a combination of the registration and login phase. First, a user enters their old password to confirm the action, following the same procedure as during login. Second, a new password is chosen, which follows the requirements as in the registration phase. After the password is successfully changed, the user can only login using the new password; the old password is invalidated.

4.2 Encoding passwords as input of a zk-SNARK

Zk-SNARKs operate on inputs over a field $\mathbb{F}$, typically $\mathbb{F}_p$, where p is an l-bit prime. However, a password is a variable-length string and thus incompatible as input of a zk-SNARK. For a zk-SNARK to receive a password as input, the password should thus be mapped to an element $e \in \mathbb{F}$, ideally without loss of its properties.

One option is to encode each character of the password string (e.g. using ASCII encoding) and use a separate input for each encoded character. While this eases string encoding and accessing individual characters in the zk-SNARK computation, it has the drawback of being inefficient due to the large circuit size and number of constraints. In addition, there is a hard limit on the maximum password length, because a zk-SNARK with n inputs can only support passwords with a length up to n characters.

Another way to encode a string is described by [KM14]. As part of their research, the authors describe a way to map a string of ASCII characters to a single integer. This has the advantage that a zk-SNARK only requires one input for the password, which results in a smaller circuit size and less constraints. Therefore it is the basis of the password encoding algorithm described below.

While the approach described by [KM14] is similar, it is specifically targeted at ASCII strings, although they mention that it can be easily adapted to other character sets such as UTF-8. In contrast, the approach proposed by us is universal for any character set. In addition, [KM14] uses several constructs that are not relevant for this research and can thus be omitted, resulting in a simplified algorithm.

We define the field $\mathbb{F}$ to be $\mathbb{F}_p$, where p is an l-bit prime. Mapping a string to an element $e \in \mathbb{F}_p$ then consists of two steps: mapping each character c_i of the string to an element $e_i \in \mathbb{Z}_b$ for a base b and aggregating each e_i into a single element $e \in \mathbb{F}_p$.

First of all, we define a character set Σ that contains all characters that can be used in a password. This character set can be ASCII, UTF-8 or any other (custom) character set. We then choose a base $b \in \mathbb{F}_p$ such that $b > n$, where $n = |\Sigma|$. The natural choice for b is $n + 1$.

Secondly, we define an injective mapping function $\phi : \Sigma \rightarrow \mathbb{Z}_b \setminus \{0\}$ that maps a character $c \in \Sigma$ to en element $e \in \mathbb{Z}_b \setminus \{0\}$. A character cannot be mapped to 0, since this term will not contribute to the aggregation step and will thus be disregarded (e.g. if $\phi(\mathtt{a}) = 0$, then `a`, `aa` and `aaa` will all be mapped to the same element 0).

We can then aggregate all e_i for $0 \leq i < k$, where k is the length of the password string, by calculating

$$e = \sum_{i=0}^{k-1} e_i \cdot b^i, \tag{4.1}$$

which results in a single element $e \in \mathbb{F}_p$ that is the unique encoding for a password. The algorithm for encoding a password is shown in Algorithm 1.

The operations can be reversed to decode the encoded string. Each e_i can be retrieved from e using

$$e_i = \lfloor \frac{e}{b^i} \rfloor \bmod b, \tag{4.2}$$

where $\lfloor \frac{a}{b} \rfloor$ means integer division. Each e_i can then be mapped to its corresponding character by reversing the mapping. The decoding algorithm is shown in Algorithm 2.

Because $e \in \mathbb{F}_p$, it is required that $e < p$. Hence, there is an implicit maximum length, depending on the base b and prime p. The maximum password length can be expressed in the following way:

$$l_{max} = \lfloor \log_b(p) \rfloor \tag{4.3}$$

The larger the character set Σ, the larger b and thus the smaller the maximum password length.

Algorithm 1 Encode password

Require: base b, mapping function ϕ, password $\hat{p}$
Ensure: encoding of the password

```
function ENCODE_PASSWORD($b, \phi, \hat{p}$)
    $e \leftarrow 0$
    for $i \leftarrow 0$ to LENGTH($\hat{p}$) $- 1$ do
        $c_i \leftarrow$ CHARAT($\hat{p}, i$)
        $e_i \leftarrow \phi(c_i)$
        $e \leftarrow e + e_i \cdot b^i$
    end for
    return $e$
end function
```

Algorithm 2 Decode password

Require: base b, mapping function ϕ, endoded password e
Ensure: decoded password

```
function DECODE_PASSWORD($b, \phi, e$)
    $\hat{p} \leftarrow$ ""
    for $i \leftarrow 0$ to $\log_b(e)$ do
        $e_i \leftarrow \lfloor \frac{e}{b^i} \rfloor \bmod b$
        $c_i \leftarrow \phi^{-1}(e_i)$
        $\hat{p} \leftarrow \hat{p} + c_i$
    end for
    return $\hat{p}$
end function
```

4.3 Encoding password policies in a zk-SNARK

Now that the password encoding scheme is defined, password policies can be expressed. Since zk-SNARKs can proof any program up to a certain size, arbitrary password policies can be enforced (up to that size). This section first describes the general approach of creating password policies (Section 4.3.1). Section 4.3.2 then illustrates the implementation of the most common password policies.

4.3.1 Creating password policies

With the password as input of the zk-SNARK, constraints can be designed such that a valid proof can only be generated if the password follows these constraints. If these constraints are encoded password policies, then it follows that if and only if the password complies to the implemented password policies, a valid proof can be generated. Hence, checking the validity of a proof affirms whether the password complies to the password policies.

A password policy receives the password as input, possibly together with other auxiliary inputs. It then proceeds with asserting certain properties of the password using constraints. If all constraints are met, the password is accepted by the policy. Otherwise, it is rejected. Next to creating individual password policies, policies can also be combined using logical operators such as AND and OR. This allows for more complex password policies.

4.3.2 Example policies

This section describes the implementation of four common password policies, namely minimum password length (Section 4.3.2.1), minimum number of characters from a subset (4.3.2.2), password not in blocklist (4.3.2.3) and substring of password not in blocklist (4.3.2.4).

4.3.2.1 Minimum password length

One of the most deployed password policies is the minimum length check. To ensure strong passwords, passwords with less than n characters are often invalidated. The exact value of n can be chosen independently by each service and is often a trade-off between security (the more characters, the more difficult the password can be guessed) and usability (the less characters, the easier to remember the password).

In order to check whether the password consists of at least n characters, it suffices to compare the encoded password to b^{n-1}, where b is the base chosen in the password encoding step. Since the encoding of a password with less than n characters is at most $\sum_{i=0}^{n-2}(b-1) \cdot b^i = b^{n-1} - 1$, a password is valid if its encoding is greater than $b^{n-1} - 1$. The corresponding algorithm is shown in Algorithm 3.

4.3.2.2 Minimum number of characters from a subset

Another often deployed password policy is checking whether a passwords contains a minimum number of characters from certain character sets. Typically used character sets are lowercase characters, uppercase characters, digits and symbols. The

Algorithm 3 Enforcement of minimum password length

Require: base b, minimum $n \geq 1$, valid password encoding e

Ensure: reject password if e contains less than n characters

```
function MinPasswordLength($b, n, e$)
    if $e < b^{n-1}$ then
        Reject($e$)
    else
        Accept($e$)
    end if
end function
```

required minimum number of characters n can be chosen independently for each character set.

Checking whether a password contains the minimum number of characters from a certain subset comes down to extracting the characters from the encoded password using Equation 4.2 and counting each character that is in the subset. The resulting sum k can then be compared to the predefined minimum number of characters n. The password is rejected if $k < n$. This algorithm is also shown in Algorithm 4.

Algorithm 4 Enforcement of minimum number of characters from a subset

Require: base b, subset s, minimum $n \geq 0$, valid password encoding e

Ensure: rejects password if e contains less than n characters from s

```
function MinCharactersFromSubset($b, s, n, e$)
    $k \leftarrow 0$
    for $i \leftarrow 0$ to PasswordLength($e$) $- 1$ do
        $c \leftarrow \lfloor \frac{e}{b^i} \rfloor \bmod b$
        if $c \in s$ then
            $k \leftarrow k + 1$
        end if
    end for
    if $k < n$ then
        Reject($e$)
    else
        Accept($e$)
    end if
end function
```

4.3.2.3 Password not in blocklist

While the previously discussed password policies restrict passwords that do not conform to certain requirements, it may still allow commonly used passwords and compromised passwords. These types of passwords are susceptible to dictionary attacks, in which an attacker uses a dictionary of known passwords to guess the password of a certain account. Disallowing these kinds of passwords can mitigate these kinds of attacks and thus greatly improve account security.

Common passwords can be collected and compiled into a blocklist, in which a user's chosen password may not occur. The straightforward way to check whether a password occurs on the blocklist is by comparing the password with each entry of the blocklist and rejecting the password if any of the entries matches the password. While this ensures that there are no false positives and false negatives, the password needs to be compared to all entries, which is not efficient for large blocklists.

A well-researched construct that can improve the efficiency is an *Approximate Membership Query Filter (AMQ-Filter)*. By inserting all passwords from the blocklist into an AMQ-Filter, the blocklist can (1) be stored more efficiently in a zk-SNARK and (2) be queried more efficiently, at the expense of introducing a false-positive probability ϵ. There are several types of AMQ-Filters with their own strengths and weaknesses, among which the Bloom filter [Blo70], cuckoo filter [FAKM14] and xor filter [GL20]. From these, the xor filter is shown to be the most space-efficient. While xor filters do not support inserting elements after construction, this does not impact its suitability for password blocklists, as these lists can be generated once and embedded in a zk-SNARK. If more passwords need to be added, the filter can be constructed from scratch again. Hence, the xor filter has been opted for.

Since zk-SNARKs operate over a field $\mathbb{F}_p$, the fingerprints of the filter must be represented as field elements. When storing each fingerprint separately as an element, the space complexity will be very large. Since the number of fingerprints stored in the filter is larger than the length of the blocklist, the space complexity would be even larger than including the original blocklist. Instead, we can use the fact that all fingerprints have a fixed size to our advantage. By fitting as many fingerprints inside an element in $\mathbb{F}_p$, the space requirements are greatly reduced. Similar to the encoding of passwords discussed in Chapter 4.2, this is done by stacking the fingerprints, with the fingerprint size as base. The encoding algorithm is shown in Algorithm 5.

In order to use a blocklist filter, several steps need to be performed. First of all, a blocklist should be created, containing passwords which are encoded as described in Section 4.2. Secondly, a suitable hash function needs to be chosen that should be efficient for use in zk-SNARKs, a so-called *SNARK-friendly hash function*. This hash function does not need to be cryptographically secure, as long as it is sufficiently

Algorithm 5 Encode xor filter

Require: xor filter f, fingerprint bit size s

Ensure: encoded xor filter

```
function EncodeXorFilter($f$, $s$)
    $fingerprintsPerElement \leftarrow \lfloor \frac{\textsc{MaxBits}}{s} \rfloor$
    $elements \leftarrow []$
    for $i \leftarrow 0$ to $\lceil \frac{\textsc{Length}(f)}{fingerprintsPerElement} \rceil$ do
        $start \leftarrow i \cdot fingerprintsPerElement$
        $end \leftarrow \textsc{min}((i+1) \cdot fingerprintsPerElement, \textsc{Length}(f))$
        $el \leftarrow 0$
        for $j \leftarrow start$ to $end - 1$ do
            $el \leftarrow el + (\textsc{Get}(f, j) << ((fingerprintsPerElement - 1 - j) * s))$
        end for
        Append($elements, el$)
    end for
    return $elements$
end function
```

independent and uniformly distributed. Examples of SNARK-friendly hash functions are MiMC7 [AGR+16], Poseidon [GKR+21], Pedersen [BHW20] and Rescue Prime [SAD20], which are all suitable to be used as hash function in the blocklist filter. To obtain three different hash functions, different seeds can be used.

After construction, the filter is encoded using Algorithm 5 and embedded in the zk-SNARK. Testing whether the blocklist filter contains a password is done as follows. First, the password is hashed with the three hash functions generated in the construction phase. While the hash indicates the index of the fingerprint in the filter, it cannot be used directly, since the filter is encoded. Instead, the index in the encoded filter can be retrieved by dividing the hash by the number of fingerprints per element. Similarly, to obtain the correct fingerprint part from the element at the found index, the hash modulo the number of fingerprints per element is calculated. After extracting them from the encoded filter, the fingerprints are XOR'ed. Finally, the fingerprint of the password is compared to the result of the XOR operations. If they are the same, then the password is probably on the blocklist and thus rejected. Otherwise, the password is accepted. This algorithm is also shown in Algorithm 6.

As already mentioned, the use of AMQ-Filters introduces a false positive rate ϵ. In the context of password blocklists, this means that some passwords will be rejected, while not present in the blocklist. The choice of fingerprint size determines the exact value of ϵ. While having small fingerprints is beneficial in terms of filter size, it incurs

Algorithm 6 Enforcement of password not in blocklist

Require: encoded xor filter f, fingerprint bit size s, hash functions h_0, h_1, h_2, valid password encoding e

Ensure: `in filter` if $e \in f$ with error probability ϵ; `not in filter` if $e \notin f$

```
function PasswordNotInBlocklist($f, s, h_0, h_1, h_2, e$)
    $fingerprintsPerElement \leftarrow \lfloor \frac{\text{MaxBits}}{s} \rfloor$
    $fp \leftarrow$ Fingerprint($e$)
    $res \leftarrow 0$
    for $i \leftarrow 0$ to $2$ do
        $hash \leftarrow h_i(e)$
        $elementNo \leftarrow \lfloor \frac{hash}{fingerprintsPerElement} \rfloor$
        $elementPart \leftarrow hash \bmod fingerprintsPerElement$
        $res \leftarrow res \oplus ((\text{Get}(f, elementNo) >> (fingerprintsPerElement - elementPart - 1) * s) \;\&\; ((1 << s) - 1))$
    end for
    if $fp = res$ then
        Reject($e$)
    else
        Accept($e$)
    end if
end function
```

a higher false positive rate. There is thus a trade-off between fingerprint size and false positive rate.

4.3.2.4 Substring of password not in blocklist

Similar to disallowing certain passwords, it can benefit security to block certain substrings. For example, passwords containing ‘password’ or ‘123’ are very likely easy to guess. By compiling a blocklist of forbidden substrings, this kind of passwords can be effectively blocked.

In contrast to the password blocklist, it is not possible to use an AMQ-Filter to efficiently check whether a substring is in the filter in the same way. This has two reasons. First of all, changing one bit in the input will result in a completely different hash. Hence, a password cannot be checked once for all substrings. Secondly, while it is possible to compare each substring of a password to a blocklist filter, this will not be efficient. Since a password of length n contains $\frac{n \cdot (n+1)}{2}$ substrings, $\frac{n \cdot (n+1)}{2}$ queries to the filter are required to check all substrings. Therefore, this approach is

Algorithm 7 Enforcement of substring of password not in blocklist

Require: base b, blocklist l, valid password encoding e

Ensure: rejects password if e contains a substring $s \in l$

```
function SubStringOfPasswordNotInBlocklist($b, l, e$)
    $passwordLength \leftarrow$ PasswordLength($e$)
    for all $i \in l$ do
        $substringLength \leftarrow$ PasswordLength($i$)
        for $j \leftarrow 0$ to $passwordLength - substringLength$ do
            $s \leftarrow \lfloor \frac{e}{b^j} \rfloor \bmod b^{substringLength}$
            if $s \in b$ then
                Reject($e$)
            end if
        end for
    end for
    Accept($e$)
end function
```

infeasible.

Instead, the approach is to compile a list of forbidden substrings, encode each substring with the encoding discussed in Section 4.2 and embed the blocklist in the zk-SNARK. Checking whether the password contains any substring then comes down to iterating over the substrings in the blocklist and comparing each substring with the password. To do this comparison, an iteration is required over all possible starting positions of that substring, and comparing each part of the password with that substring. If any of the substrings matches, the password is rejected. Otherwise, it is accepted. This algorihm is also shown in Algorithm 7.

While this approach has the advantage of not introducing any false positives, it has as drawback that it is less efficient than using AMQ-Filters. Hence, for performance reasons, the substring blocklist cannot be very large.

4.4 Protocol

Using the results from Sections 4.2 and 4.3, the protocol can be built. The protocol achieves zero-knowledge of the password, while still being able to enforce password policies. This section describes the different phases of the protocol, which are `Setup`, `Registration`, `Login` and `Change password`.

4.4.1 Setup

Before the protocol can be used, a setup is required. This setup is executed solely by the server and comprises setting up the cryptographic primitives and generating cryptographic keys. More specifically, the setup entails setting up the zk-SNARK Common Reference String (CRS), setting up SAVER and generating the SAVER keys. The key generation results in three different keys, namely the public key PK, secret key SK and verification key VK. PK is required by the client to generate proofs and is thus public. SK and VK are required to decrypt and verify ciphertexts respectively and stay on the server. The CRS and SAVER setup is shown in Algorithm 8. The SAVER key generation is shown in Algorithm 9.

4.4.2 Registration

During the registration phase, a user chooses a username and password. In order to prove to a server that the password complies to the password policies, a zk-SNARK is used. This zk-SNARK takes as input the password and a salt, and computes and outputs the password hash. Of these inputs and outputs, the salt and hash should

Algorithm 8 SAVER setup

```
Require: relation $rel$
Ensure: Common Reference String $CRS$

function SETUP($rel$)
    $\hat{CRS} \leftarrow \Pi_{\text{snark}}.\text{SETUP}(rel)$
    $CRS \leftarrow \hat{CRS} \cup \{G^{-\gamma}\}$
    return $CRS$
end function
```

Algorithm 9 SAVER key generation

```
Require: $CRS$
Ensure: SAVER keys

function KEYGEN($CRS$)
    $s, v, t_0, t_1, \rho \xleftarrow{\$} \mathbb{Z}_p^*$
    $PK \leftarrow (G^{\delta}, G^{\delta s}, {G_1}^{t_1}, H^{t_0}, H^{t_1}, G^{\delta(t_0+t_1 s)}, G^{-\gamma(1+s)})$
    $SK \leftarrow \rho$
    $VK \leftarrow (H^{\rho}, H^{sv}, H^{\rho v})$
    return $(PK, SK, VK)$
end function
```

be public, as the salt should be stored by the server and the hash is to be encrypted using SAVER. In contrast, the password is a private input and thus never exposed to the server.

Ideally, the password hashing function should be cryptographically secure and memory-hard, such that bruteforcing passwords and computing hashes is slow. This ensures that even if an adversary gets hold of the SAVER private key, finding the password is difficult. Apart from computing the password hash, the zk-SNARK enforces password policies, implemented as discussed in Section 4.3. A valid zk-SNARK proof can only be generated if the password complies to the implemented password policies. Hence, it suffices to verify the zk-SNARK proof to check whether the password complies to the chosen password policies.

To authenticate a user after registration, some derivation of the password needs to be stored at the server. For this, we use SAVER to encrypt the output of the zk-SNARK, which is the password hash. This encryption is sent alongside the password proof to the server, which validates them both and stores the encryption with the username and salt.

The interaction flow between the client and the server is shown in Figure 4.1. The procedure of creating a proof and the corresponding encryption is shown in Algorithm 10. Finally, the verification algorithm is shown in Algorithm 11.

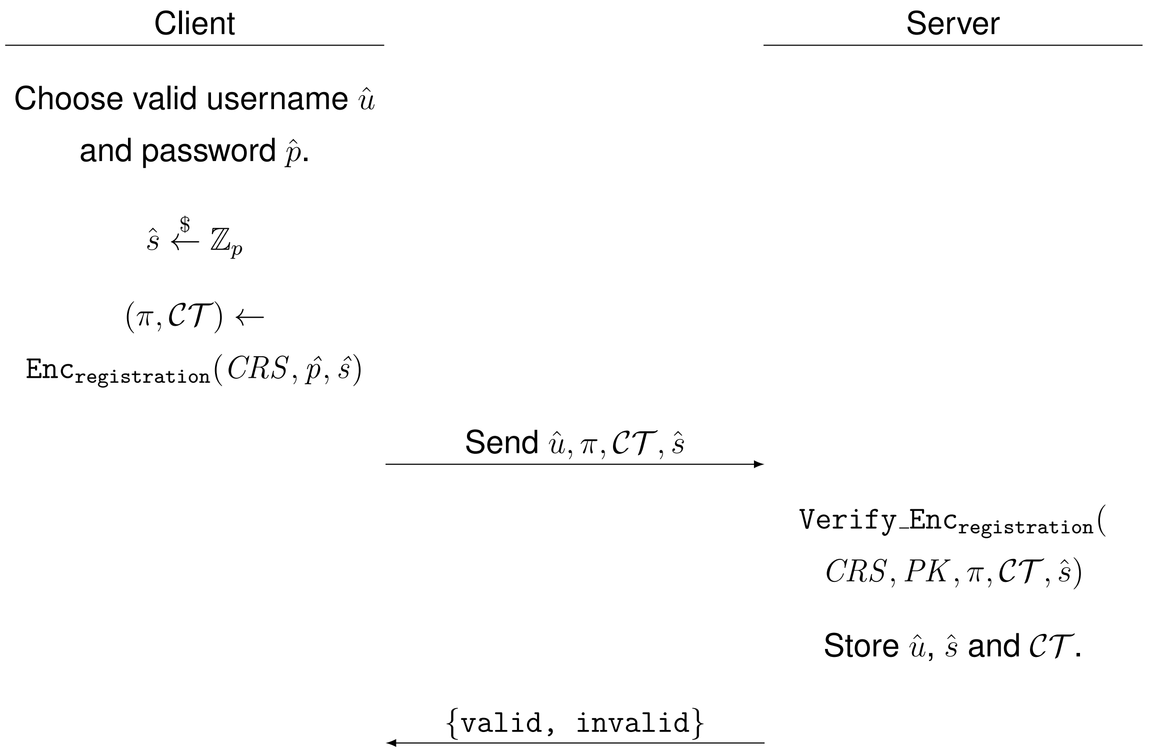

Client
Server
Choose valid username $\hat{u}$ and password $\hat{p}$.
$\hat{s} \xleftarrow{\$} \mathbb{Z}_p$
$(\pi, \mathcal{CT}) \leftarrow$ $\texttt{Enc}_{\texttt{registration}}(CRS, \hat{p}, \hat{s})$
Send $\hat{u}, \pi, \mathcal{CT}, \hat{s}$
$\texttt{Verify\_Enc}_{\texttt{registration}}($ $CRS, PK, \pi, \mathcal{CT}, \hat{s})$
Store $\hat{u}$, $\hat{s}$ and $\mathcal{CT}$.
$\{\texttt{valid, invalid}\}$

Figure 4.1: Registration interaction flow

Algorithm 10 Enc$_{\text{registration}}$

Require: CRS, SAVER public key PK, password $\hat{p}$, salt $\hat{s}$
Ensure: valid proof and password hash encryption

```
function Enc_REGISTRATION($CRS, PK, \hat{p}, \hat{s}$)
    let $PK = (X_0, X_1, Y, Z_0, Z_1, P_1, P_2)$
    $r \xleftarrow{\$} \mathbb{Z}_p^*$
    $(\hat{h}, \hat{\pi} = (A, B, C)) \leftarrow \Pi_{\text{snark}}.\text{PROVE}(CRS, \hat{p}, \hat{s})$
    $\pi \leftarrow (A, B, C \cdot {P_2}^r)$
    $\mathcal{CT} \leftarrow ({X_0}^r, {X_1}^r {G_1}^{\hat{h}}, \psi = {P_1}^r Y^{\hat{h}})$
    return $(\pi, \mathcal{CT})$
end function
```

Algorithm 11 Verify_Enc$_{\text{registration}}$

Require: CRS, SAVER public key PK, proof π, ciphertext $\mathcal{CT}$, salt $\hat{s}$
Ensure: succeeds if and only if $\mathcal{CT}$ is a valid ciphertext belonging to π and π is a valid zk-SNARK proof; fails otherwise

```
function VERIFY_Enc_REGISTRATION($CRS, PK, \pi, \mathcal{CT}, \hat{s}$)
    parse $\pi = (A, B, C)$ and $\mathcal{CT} = (c_0, c_1, \psi)$
    let $PK = (X_0, X_1, Y, Z_0, Z_1, P_1, P_2)$
    assert $e(c_0, Z_0) \cdot e(c_1, Z_1) = e(\psi, H)$
    assert $e(A, B) = e(G^\alpha, H^\beta) \cdot e(c_0 \cdot c_1 \cdot {G_2}^{\hat{s}}, H^\gamma) \cdot e(C, H^\delta)$
end function
```

4.4.3 Login

After a user is registered, they can subsequently log in to the server using the same credentials. Because the password has already been proven to comply with the policies in the registration phase, a zk-SNARK proof does not provide additional security. Instead, a similar approach to SAVER is taken to arrive at an encryption, which can subsequently be checked by the server.

When a client wants to log in with their credentials, they first request the salt that was used in the registration phase. Since salts are public information, those can be returned on request via an API call or stored in a public directory. Using the salt, the client hashes the password with the same hashing function $\mathcal{H}_p$ as used in the zk-SNARK, mirroring the hash calculation of the zk-SNARK. Secondly, it uses the same SAVER parameters and keys as in the registration phase to encrypt the password hash. Because SAVER's encryption scheme is probabilistic, the ciphertext will be different from the ciphertext stored at the server.

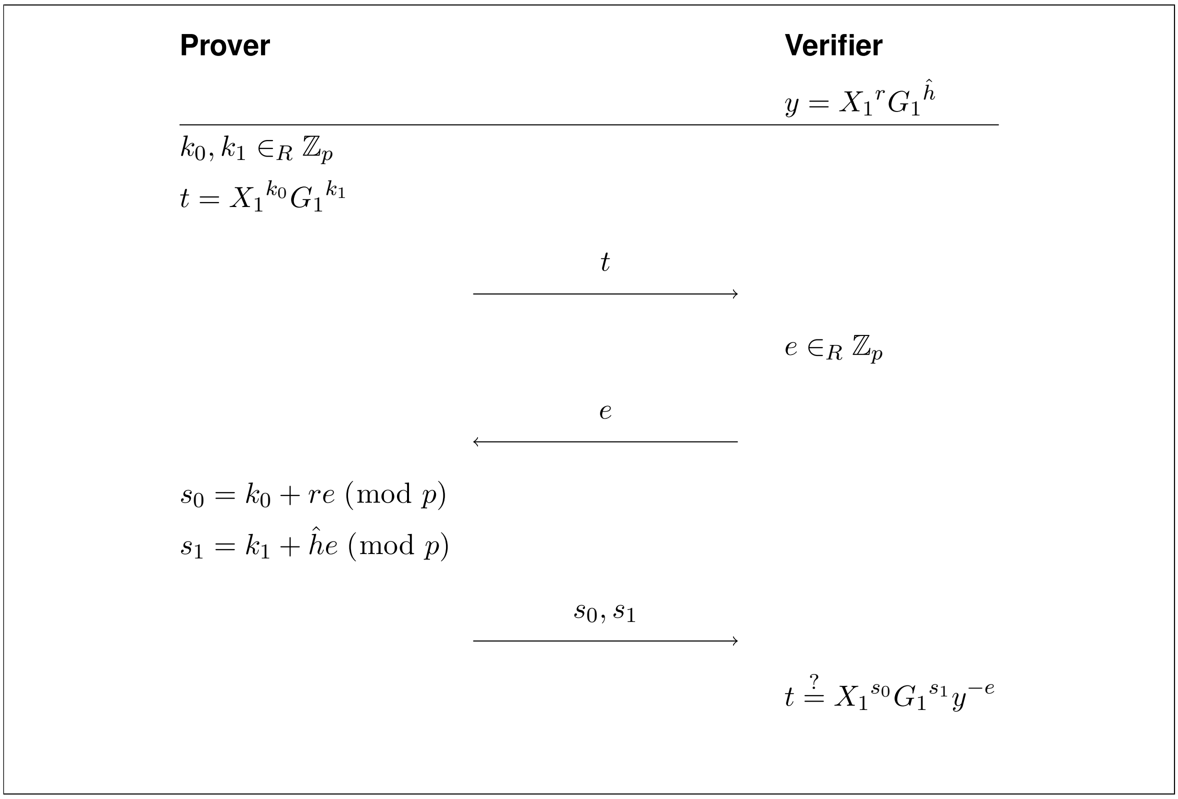

Prover
Verifier
$y = {X_1}^r {G_1}^{\hat{h}}$
$k_0, k_1 \in_R \mathbb{Z}_p$
$t = {X_1}^{k_0} {G_1}^{k_1}$
$\xrightarrow{t}$
$e \in_R \mathbb{Z}_p$
$\xleftarrow{e}$
$s_0 = k_0 + re \pmod{p}$
$s_1 = k_1 + \hat{h}e \pmod{p}$
$\xrightarrow{s_0, s_1}$
$t \overset{?}{=} {X_1}^{s_0} {G_1}^{s_1} y^{-e}$

Figure 4.2: Sigma protocol proving knowledge of the exponents in c_1

To prevent adversaries from using the ciphertext stored at the server during login, a zero-knowledge proof is added to the ciphertext. This zero-knowledge proof is a sigma protocol, proving the knowledge of the exponents of the second item of the ciphertext, r and $\hat{h}$ in ${X_1}^r {G_1}^{\hat{h}}$, made non-interactive using the Fiat-Shamir heuristic [FS87]. Because of the structure of c_1, this is similar to proving knowledge of the exponents of a Pedersen commitment [Ped92]. The sigma protocol is shown in Figure 4.2. After applying the Fiat-Shamir heuristic, the non-interactive zero-knowledge proof consists of the triple $\varphi = (\varphi_{Co}, \varphi_{Ch}, \varphi_{Re})$:

$$\begin{aligned}
\varphi_{Co} &= ({X_1}^{k_0} {G_1}^{k_1}) \\
\varphi_{Ch} &= \mathcal{H}_\varphi(X_1, G_1, y, \varphi_{Co}) \\
\varphi_{Re} &= (k_0 + r \cdot \varphi_{Ch} \pmod{p},\ k_1 + \hat{h} \cdot \varphi_{Ch} \pmod{p})
\end{aligned}$$

where $\mathcal{H}_\varphi$ is a cryptographic hash function. Because of this non-interactive zero-knowledge proof, randomizing the SAVER encryption will not result in a valid proof, unless the exponents are known, with which a new proof can be generated.

The username $\hat{u}$ and the ciphertext $\mathcal{CT}$ are then sent to the server. First, the server checks whether $\mathcal{CT}$ is well-formed. Secondly, it verifies the correctness of the included non-interactive zero-knowledge proof, by checking whether

$$\mathcal{H}_\varphi(X_1, G_1, y, {X_1}^{s_0} {G_1}^{s_1} y^{-\varphi_{Ch}}) \overset{?}{=} \varphi_{Ch}.$$

If these two checks succeed, the ciphertext is well-formed and valid. However, this does not mean that the credentials are correct. For this, the server retrieves the stored ciphertext $\mathcal{CT}'$ using $\hat{u}$. Now, there are two ciphertexts $\mathcal{CT}$ and $\mathcal{CT}'$ that need to be compared to find whether the same $\hat{h}$ was encrypted. This can be done in two ways. First of all, both ciphertexts can be decrypted similar to SAVER to obtain $e(G_1, V_2)^{\hat{h}}$ and $e(G_1, V_2)^{\hat{h}'}$, which can directly be compared for equality.

The second approach is to exploit the homomorphic property of SAVER. Instead of decrypting both ciphertexts, the ciphertexts are combined using the division operator:

$$\begin{aligned}\mathcal{CT}'' &= (\frac{c_0}{{c_0}'}, \frac{c_1}{{c_1}'}) \\ &= (\frac{{X_0}^r}{{X_0}^{r'}}, \frac{{X_1}^r {G_1}^{\hat{h}}}{{X_1}^{r'} {G_1}^{\hat{h}'}}) \\ &= ({X_0}^{r-r'}, {X_1}^{r-r'} {G_1}^{\hat{h}-\hat{h}'})\end{aligned}$$

after which $\mathcal{CT}''$ can be decrypted. It can then be checked whether the result is equal to 1, which indicates successful authentication. After all, if we have $\hat{h} = \hat{h}'$, then we find

$$\begin{aligned}\mathcal{CT}'' &= ({X_0}^{r-r'}, {X_1}^{r-r'} {G_1}^{\hat{h}-\hat{h}}) \\ &= ({X_0}^{r-r'}, {X_1}^{r-r'} {G_1}^{0})\end{aligned}$$

and the decryption will yield $e(G_1, V_2)^0 = 1$.

Each decryption requires two pairings and one division, resulting in four pairings and two divisions for the first approach. In contrast, the second approach requires only one decryption and two divisions, resulting in two pairings and three divisions. Since a division operation is more efficient than a pairing, the second approach has a better performance and is thus favored.

Algorithm 12 shows the algorithm for creating the encryption. For the server side, algorithm 13 shows the verification procedure and algorithm 14 shows the algorithm for comparing the two ciphertexts. Finally, the complete login interaction flow is shown in Figure 4.3.

4.4.4 Change password

The last part of the protocol is *change password* and is a combination of the registration and login phases. First, to confirm the action, the client performs the same procedure as during login using their current password. Secondly, a zk-SNARK proof is created in a similar way as during registration, using their newly chosen password. These two ciphertexts and proofs are sent to the server, which checks

Algorithm 12 Enc$_{\text{login}}$

Require: CRS, SAVER public key PK, password $\hat{p}$, salt $\hat{s}$

Ensure: valid password hash encryption

```
function ENC_LOGIN($CRS, PK, \hat{p}, \hat{s}$)
    let $PK = (X_0, X_1, Y, Z_0, Z_1, P_1, P_2)$
    $\hat{h} \leftarrow \mathcal{H}_p(\hat{p}; \hat{s})$
    $r, k_0, k_1 \xleftarrow{\$} \mathbb{Z}_p^*$
    $\hat{\mathcal{CT}} = (c_0, c_1, \psi) \leftarrow (X_0{}^r, X_1{}^r G_1{}^{\hat{h}}, \psi = P_1{}^r Y^{\hat{h}})$
    $\varphi_{Co} \leftarrow X_1{}^{k_0} G_1{}^{k_1}$
    $\varphi_{Ch} \leftarrow \mathcal{H}_\varphi(X_1, G_1, c_1, \varphi_{Co})$
    $\varphi_{Re} \leftarrow (k_0 + r \cdot \varphi_{Ch}, k_1 + \hat{h} \cdot \varphi_{Ch})$
    $\mathcal{CT} \leftarrow (c_0, c_1, \psi, \varphi = (\varphi_{Co}, \varphi_{Ch}, \varphi_{Re}))$
    return $\mathcal{CT}$
end function
```

Algorithm 13 Verify_Enc$_{\text{login}}$

Require: CRS, SAVER public key PK, ciphertext $\mathcal{CT}$

Ensure: succeeds if and only if $\mathcal{CT}$ is a valid ciphertext; fails otherwise

```
function VERIFY_ENC_LOGIN($CRS, PK, \mathcal{CT}$)
    parse $\mathcal{CT} = (c_0, c_1, \psi, \varphi = (\varphi_{Co}, \varphi_{Ch}, \varphi_{Re} = (s_0, s_1)))$
    let $PK = (X_0, X_1, Y, Z_0, Z_1, P_1, P_2)$
    assert $e(c_0, Z_0) \cdot e(c_1, Z_1) = e(\psi, H)$
    assert $\mathcal{H}_\varphi(X_1, G_1, c_1, X_1{}^{s_0} G_1{}^{s_1} c_1{}^{-\varphi_{Ch}}) = \varphi_{Ch}$
end function
```

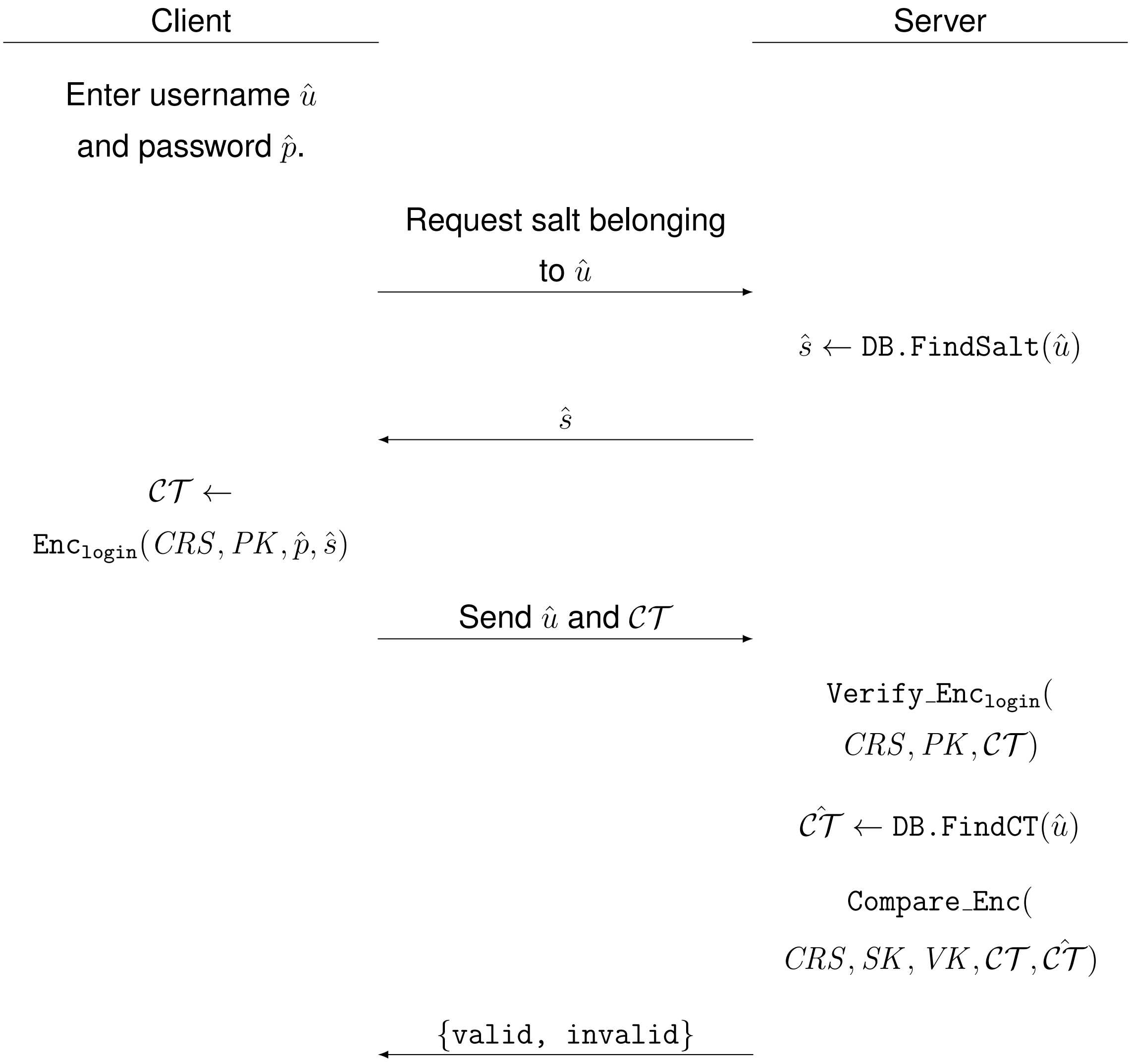

Client
Server
Enter username $\hat{u}$
and password $\hat{p}$.
Request salt belonging
to $\hat{u}$
$\hat{s} \leftarrow \texttt{DB.FindSalt}(\hat{u})$
$\hat{s}$
$\mathcal{CT} \leftarrow$
$\texttt{Enc}_{\texttt{login}}(CRS, PK, \hat{p}, \hat{s})$
Send $\hat{u}$ and $\mathcal{CT}$
$\texttt{Verify\_Enc}_{\texttt{login}}($
$CRS, PK, \mathcal{CT})$
$\hat{\mathcal{CT}} \leftarrow \texttt{DB.FindCT}(\hat{u})$
$\texttt{Compare\_Enc}($
$CRS, SK, VK, \mathcal{CT}, \hat{\mathcal{CT}})$
$\{\texttt{valid, invalid}\}$

Figure 4.3: Login interaction flow

Algorithm 14 Compare_Enc

Require: CRS, SAVER private key SK, SAVER verification key VK, first ciphertext $\mathcal{CT}$, second ciphertext $\mathcal{CT}'$

Ensure: succeeds if the encrypted password hashes are equal; fails otherwise

```
function COMPARE_ENC($CRS, SK, VK, \mathcal{CT}, \mathcal{CT}'$)
    parse $\mathcal{CT} = (c_0, c_1, \psi, \varphi)$ and $\mathcal{CT}' = (c_0', c_1', \psi', \varphi')$
    $\hat{\mathcal{CT}} \leftarrow (\frac{c_0}{c_0'}, \frac{c_1}{c_1'})$
    $m \leftarrow$ DEC($CRS, SK, VK, \hat{\mathcal{CT}}$)
    assert $m = 1$
end function

function DEC($CRS, SK, VK, \mathcal{CT}$)
    parse $sk = \rho$, $VK = (V_0, V_1, V_2)$ and $\mathcal{CT} = (c_0, c_1)$
    return $\frac{e(c_1, V_2)}{e(c_0, V_1)^\rho}$
end function
```

them similarly as in the registration and login phase. Only if both ciphertexts and proofs are correct, the stored ciphertext is replaced with the new one, invalidating the old one. This interaction flow is shown in Figure 4.4.

4.5 Protecting against replay attacks

Because the protocol is non-interactive, replay attacks are a present threat. Replay attacks are performed by capturing valid messages and retransmitting them, which may lead to tricking the server into accepting the message as if it was original. In the case of our proposed protocol, this means that if an attacker is able to capture a valid ciphertext, they can use that ciphertext to log in, even if they do not know the password.

Preventing replay attacks in our protocol is only needed for the login and change password phases. After all, if a proof in the registration phase is replayed, a new account is created and hence no additional gain can be obtained in comparison with creating an original proof.

For the other two phases, an extra freshness check on the server side can thwart replay attacks. This check is linked to the zero-knowledge proof φ, which is included in the ciphertext. Because k_0 and k_1 are chosen randomly for each proof, the commitment $\varphi_{Co} = {X_1}^{k_0}{G_1}^{k_1}$ has a very high probability to be unique. Hence, all previously received commitments can be stored by the server, possibly in a space-

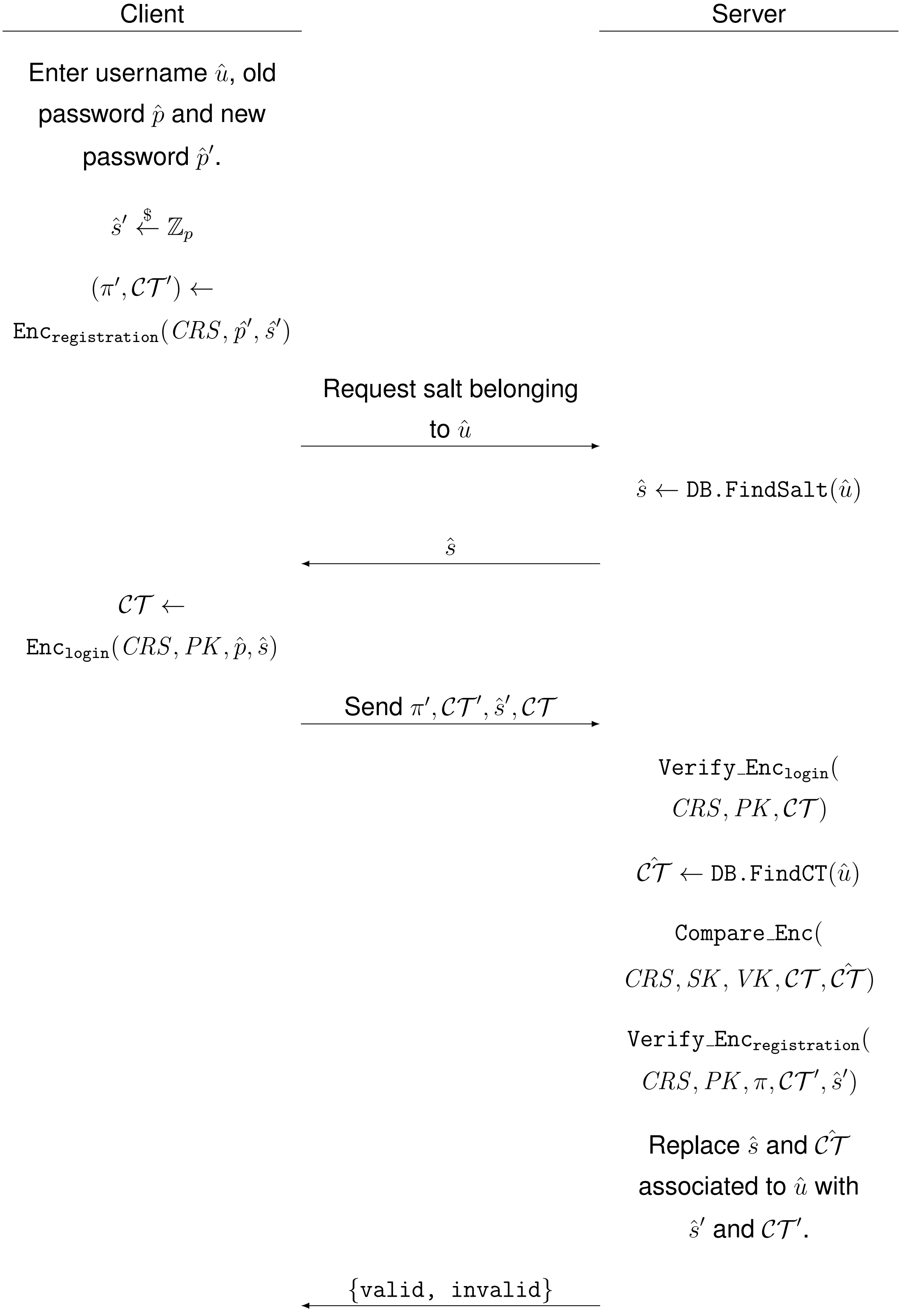

Client
Server
Enter username $\hat{u}$, old password $\hat{p}$ and new password $\hat{p}'$.
$\hat{s}' \xleftarrow{\$} \mathbb{Z}_p$
$(\pi', \mathcal{CT}') \leftarrow \texttt{Enc}_{\texttt{registration}}(CRS, \hat{p}', \hat{s}')$
Request salt belonging to $\hat{u}$
$\hat{s} \leftarrow \texttt{DB.FindSalt}(\hat{u})$
$\hat{s}$
$\mathcal{CT} \leftarrow \texttt{Enc}_{\texttt{login}}(CRS, PK, \hat{p}, \hat{s})$
Send $\pi', \mathcal{CT}', \hat{s}', \mathcal{CT}$
$\texttt{Verify\_Enc}_{\texttt{login}}(CRS, PK, \mathcal{CT})$
$\hat{\mathcal{CT}} \leftarrow \texttt{DB.FindCT}(\hat{u})$
$\texttt{Compare\_Enc}(CRS, SK, VK, \mathcal{CT}, \hat{\mathcal{CT}})$
$\texttt{Verify\_Enc}_{\texttt{registration}}(CRS, PK, \pi, \mathcal{CT}', \hat{s}')$
Replace $\hat{s}$ and $\hat{\mathcal{CT}}$ associated to $\hat{u}$ with $\hat{s}'$ and $\mathcal{CT}'$.
$\{\texttt{valid, invalid}\}$

Figure 4.4: Change password interaction flow

Algorithm 15 Verify_Freshness

Require: commitment collection Φ, ciphertext $\mathcal{CT}$

Ensure: succeeds if and only if $\mathcal{CT}$ is a fresh ciphertext; fails otherwise

```
function VERIFY_FRESHNESS($\Phi, \mathcal{CT}$)
    parse $\mathcal{CT} = (c_0, c_1, \psi, \varphi = (\varphi_{Co}, \varphi_{Ch}, \varphi_{Re}))$
    assert $\varphi_{Co} \notin \Phi$
    $\Phi \leftarrow \Phi \cup \varphi_{Co}$
end function
```

efficient way such as AMQ-Filters, and be used as a blocklist for future commitments. This disallows re-use of the zero-knowledge proof φ, which in turn disallows re-using the whole ciphertext $\mathcal{CT}$. The freshness check is shown in Algorithm 15 and should be performed by the server after Verify_Enc$_{\text{login}}$.

4.6 Security

In this section, the security of the protocol is discussed on the basis of the security assumptions and the security of its components. These components are the zk-SNARK in the registration phase, SAVER and the zero-knowledge proof of the login phase.

We model the server as an honest-but-curious adversary. This is because it needs to be trusted to generate correct keys and throw away the toxic waste of the zk-SNARK setup. However, it might still want to learn as much as possible.

The zk-SNARK system used in the registration phase, [Gro16], has been shown to be secure in the generic bilinear group model (GBGM) [Sho97, Gro16]. With small modifications, [ABLZ17] shows that the zk-SNARK system is secure in the subversion generic bilinear group model (Sub-GBGM), which is stronger than GBGM. Since no modifications are done to the zk-SNARK system, the GBGM security also holds in the proposed protocol.

SAVER has been proven to be IND-CPA secure [LCKO19]. In addition, the authors prove the soundness of the ciphertext and zk-SNARK resulting from the Enc phase. Since we do not alter the encryption scheme (except for adding an additional element in the form of a non-interactive zero-knowledge proof), the same security holds in the proposed protocol.

The zero-knowledge proof added to the ciphertext in the login phase is a sigma protocol proving knowledge of a Pedersen commitment, made non-interactive using the Fiat-Shamir heuristic [FS87]. Its security thus consists of two parts, namely that of the Pedersen commitment and of the non-interactive sigma protocol.

A Pedersen commitment has been shown to be information theoretically hiding and computationally binding. Hence, it is impossible for an adversary to find the committed values purely based on its commitment. Finding different values which result in the same commitment is possible, but is considered hard.

The interactive sigma protocol shown in Figure 4.2 can be shown to be secure based on three requirements, namely *completeness*, *soundness* and *zero-knowledge* (see Section 2.3), provided that the verifier is honest. We now provide a proof that the interactive sigma protocol proving knowledge of a Pedersen commitment is a secure zero-knowledge protocol. When using the Fiat-Shamir heuristic to make the sigma protocol non-interactive, the result is proven to be secure in the random oracle model [PS96].

Theorem 1. *The interactive sigma protocol shown in Figure 4.2 is a secure zero-knowledge procotol.*

Proof. We prove this by showing that the sigma protocol conforms to the requirements of a secure zero-knowledge protocol:

Completeness. If the prover knows r and $\hat{h}$, then the verifier always accepts the proof:

$$\begin{aligned} t &= {X_1}^{s_0}{G_1}^{s_1}y^{-e} \\ &= {X_1}^{k_0+re}{G_1}^{k_1+\hat{h}e}{X_1}^{-re}{G_1}^{-\hat{h}e} \\ &= {X_1}^{k_0+re-re}{G_1}^{k_1+\hat{h}e-\hat{h}e} \\ &\overset{\checkmark}{=} {X_1}^{k_0}{G_1}^{k_1} \end{aligned}$$

Soundness. To prove soundness, we show that the protocol is *special-sound*, which implies *soundness*. We do this by showing that given two protocol runs with the same commitment but different challenges ((t, e, s_0, s_1) and (t, e', ${s_0}'$, ${s_1}'$), where $e \neq e'$), an adversary can recover the committed values:

$$\begin{aligned} {X_1}^{s_0}{G_1}^{s_1}y^{-e} &= {X_1}^{{s_0}'}{G_1}^{{s_1}'}y^{-e'} \\ {X_1}^{s_0}{G_1}^{s_1}{X_1}^{-re}{G_1}^{-\hat{h}e} &= {X_1}^{{s_0}'}{G_1}^{{s_1}'}{X_1}^{-re}{G_1}^{-\hat{h}e'} \\ {X_1}^{s_0-re}{G_1}^{s_1-\hat{h}e} &= {X_1}^{{s_0}'-re'}{G_1}^{{s_1}'-\hat{h}e'} \\ s_0 - re = {s_0}' - re' &\ \wedge\ s_1 - \hat{h}e = {s_1}' - \hat{h}e' \\ r = \frac{s_0 - {s_0}'}{e - e'} &\ \wedge\ \hat{h} = \frac{s_1 - {s_1}'}{e - e'} \end{aligned}$$

Zero-knowledge. We prove the zero-knowledge requirement by showing that a real transcript and a simulation are indistinguishable .The distribution of transcripts generated by the real protocol is:

$$\{(t, e, s_0, s_1) : k_0, k_1 \in_R \mathbb{Z}_p; e \in_R \mathbb{Z}_p; t = {X_1}^{k_0}{G_1}^{k_1};\\ s_0 = k_0 + re \pmod p; s_1 = k_1 + \hat{h}e \pmod p; \}$$

The distribution of accepting transcripts that the simulator can generate is:

$$\{(t, e, s_0, s_1) : s_0, s_1 \in_R \mathbb{Z}_p; e \in_R \mathbb{Z}_p; t = {X_1}^{s_0}{G_1}^{s_1}y^{-e}\}$$

These two distributions are identical, and each transcript occurs with equal probability. Hence, the real transcript and simulation are indistinguishable. □

Chapter 5

Evaluation

In order to show the feasibility of the proposed protocol, an implementation has been built. Section 5.1 provides details about this implementation. Section 5.2 then proceeds by showing benchmarks of the implementation to estimate its performance.

5.1 Implementation

To create and verifiy zk-SNARK proofs, the Node.js library *snarkjs* [ide] is used, which relies on Circom [BmIMt+22] for designing circuits. We extended the snarkjs library to support SAVER's setup, key generation, encryption, verify encryption and decryption operations[1]. This extended library is then used in another Node.js library *schnapsjs*[2], which implements the protocol functions and exposes them via an API. The API consists of the following functions:

- Registration:
  - CREATEPROOF
  - VERIFYPROOF
- Login:
  - CREATEENCRYPTION
  - VERIFYENCRYPTION
  - COMPAREENCRYPTIONS

Next to these libraries, a Rust program has been developed to create password blocklists as described in Section 4.3.2.3[3]. It takes a list of passwords as input, and

[1] https://github.com/Matthiti/snarkjs

[2] https://gitlab.com/Matthiti/schnapsjs

[3] https://gitlab.com/Matthiti/zk-snark-password-blocklist-encoder

outputs the generated parameters and encoded blocklist, such that it can be embedded in a zk-SNARK. Finally, a demo application has been developed, showcasing the real-world use of the schnapsjs library, consisting of a server and a client[4].

5.2 Performance

As described in Section 4.2, a password should first be encoded before it can be used in a zk-SNARK. The character set Σ used in these benchmarks consists of all printable ASCII characters, which yields the following mapping function:

$$\phi(c) = \begin{cases} \perp & \text{if } \texttt{ASCII}(c) < 32 \\ \texttt{ASCII}(c) - 31 & \text{if } 32 \leq \texttt{ASCII}(c) \leq 126 \\ \perp & \text{if } \texttt{ASCII}(c) > 126 \end{cases} \tag{5.1}$$

Since $n = |\Sigma| = 95$, we use the base $b = n + 1 = 96$. The complete mapping is shown in Table 5.1.

To obtain a representative set of passwords for the password blocklists, the xato.net password list is used, which contains around five million unique passwords [Mie19]. MiMC7 [AGR$^+$16] is used as the hash function of the xor filter, with different seeds to obtain distinct hash functions.

All benchmarks have been performed on a 2020 MacBook Pro 13” with a 2.0-GHz quad-core Intel Core i5-1038NG7 processor and 16 GB of RAM.

5.2.1 schnapsjs

The performance of the schnapsjs library can be measured in terms of execution time per function. The functions are parameterized by zk-SNARK circuit and thus by choice of password policies, for which the following scenarios have been chosen:

(A) Minimum password length (8)

(B) Minimum password length (8) + lowercase characters (1) + uppercase characters (1) + digits (1) + symbol (1)

(C) Minimum password length (8) + blocklist with 10,000 passwords with fingerprint size 8

(D) Minimum password length (8) + blocklist with 100,000 passwords with fingerprint size 8

[4] `https://gitlab.com/Matthiti/zk-schnaps-demo`

Table 5.1: Mapping resulting from Equation 5.1

Character	Value	Character	Value	Character	Value
␣	1	@	33	`	65
!	2	A	34	a	66
”	3	B	35	b	67
#	4	C	36	c	68
$	5	D	37	d	69
%	6	E	38	e	70
&	7	F	39	f	71
’	8	G	40	g	72
(	9	H	41	h	73
)	10	I	42	i	74
*	11	J	43	j	75
+	12	K	44	k	76
,	13	L	45	l	77
-	14	M	46	m	78
.	15	N	47	n	79
/	16	O	48	o	80
0	17	P	49	p	81
1	18	Q	50	q	82
2	19	R	51	r	83
3	20	S	52	s	84
4	21	T	53	t	85
5	22	U	54	u	86
6	23	V	55	v	87
7	24	W	56	w	88
8	25	X	57	x	89
9	26	Y	58	y	90
:	27	Z	59	z	91
;	28	[	60	{	92
<	29	\	61	—	93
=	30	]	62	}	94
>	31	ˆ	63	˜	95
?	32	_	64		

(E) Minimum password length (8) + substring blocklist with 100 substrings

(F) Minimum password length (8) + lowercase characters (1) + uppercase characters (1) + digits (1) + symbol (1) + blocklist with 100,000 passwords with fingerprint size 8 + substring blocklist with 100 substrings

Node.js version 16.14.0 was used for the benchmark. Each configuration has been run ten times for each function, after which the average time was noted down. The password used in all scenarios is '$N@RK$@r3@w3$0m3!', which passes all password policies described above. The results are shown in Table 5.2. From this table, it can be seen that only the proving time for the registration phase is significantly affected by the choice of password policies. All other functions run well under one second, demonstrating practical performance.

5.2.2 Password blocklist

Evaluating the performance of the password blocklist xor filter consists of three measurements: setup time, number of zk-SNARK constraints and false positive percentage. These are parameterized by fingerprint size and blocklist length. To obtain the false positive percentage, the same 100,000 passwords not present in the filter were tested for each configuration.

Rust version 1.60.0 was used for the benchmark. Each configuration has been run ten times, after which the average times and false positive percentage was noted down. The results are shown in Table 5.3. From this table it can be seen that the setup time and encoding time increase linearly with the blocklist length. In addition, it is apparent that there is a trade-off between blocklist length and fingerprint size in terms of the number of generated constraints. With a small blocklist length, a larger fingerprint size results in less constraints. This is due to the fact that retrieving small fingerprints from the filter is more expensive than traversing the filter for small blocklists. On the other hand, with larger blocklists, a smaller fingerprint is more efficient, resulting from expensive filter traversals.

Table 5.2: Performance of the schnapsjs library functions in terms of execution time

Function	**Scenario**	**Time (s)**
REGISTER.CREATEPROOF	A	1.987
	B	2.150
	C	2.918
	D	4.345
	E	2.481
	F	4.823
REGISTER.VERIFYPROOF	A	0.903
	B	0.879
	C	0.934
	D	0.932
	E	0.886
	F	0.918
LOGIN.CREATEENCRYPTION	A	0.885
	B	0.875
	C	0.866
	D	0.849
	E	0.860
	F	0.856
LOGIN.VERIFYENCRYPTION	A	0.878
	B	0.868
	C	0.847
	D	0.861
	E	0.840
	F	0.882
LOGIN.COMPAREENCRYPTIONS	A	0.886
	B	0.870
	C	0.884
	D	0.867
	E	0.867
	F	0.857

Table 5.3: Performance of the blocklist filter in terms of setup time, encoding time, false positive percentage and number of constraints

Blocklist length	Fingerprint size	Setup time (s)	Encoding time (s)	False positive percentage	Number of constraints
1,000	20	0.122	0.000	0.000%	8,201
1,000	16	0.095	0.000	0.002%	8,726
1,000	12	0.094	0.000	0.026%	10,832
1,000	8	0.093	0.000	0.389%	14,126
1,000	7	0.106	0.000	0.779%	16,172
1,000	6	0.098	0.000	1.582%	18,419
1,000	5	0.093	0.000	3.102%	21,293
1,000	4	0.100	0.000	6.259%	26,414
1,000	3	0.107	0.000	12.457%	34,484
1,000	2	0.116	0.000	24.922%	50,687
1,000	1	0.108	0.000	49.963%	99,404
10,000	20	0.983	0.001	0.000%	13,733
10,000	16	1.037	0.001	0.002%	13,154
10,000	12	1.096	0.001	0.025%	13,994
10,000	8	0.993	0.002	0.389%	16,268
10,000	7	0.968	0.001	0.779%	18,014
10,000	6	0.966	0.001	1.569%	19,997
10,000	5	0.971	0.001	3.115%	22,619
10,000	4	0.894	0.001	6.206%	27,464
10,000	3	0.961	0.001	12.514%	35,270
10,000	2	0.982	0.001	24.991%	51,209
10,000	1	1.097	0.001	50.035%	99,662
50,000	20	4.606	0.006	0.000%	38,333
50,000	16	4.551	0.006	0.002%	32,834
50,000	12	4.486	0.006	0.025%	28,052
50,000	8	4.563	0.006	0.396%	25,790

50,000	7	4.507	0.006	0.775%	26,216
50,000	6	4.526	0.006	1.540%	27,029
50,000	5	4.495	0.006	3.139%	28,523
50,000	4	4.517	0.005	6.226%	32,150
50,000	3	4.463	0.005	12.537%	38,786
50,000	2	4.573	0.005	24.959%	53,555
50,000	1	4.491	0.003	49.956%	100,838
100,000	20	9.243	0.013	0.000%	69,083
100,000	16	9.230	0.014	0.001%	57,434
100,000	12	9.098	0.011	0.023%	45,620
100,000	8	9.259	0.013	0.397%	37,694
100,000	7	9.169	0.012	0.768%	36,464
100,000	6	9.061	0.012	1.558%	35,813
100,000	5	9.133	0.012	3.146%	35,903
100,000	4	8.958	0.011	6.275%	38,006
100,000	3	9.007	0.010	12.544%	43,178
100,000	2	9.098	0.010	24.984%	56,483
100,000	1	9.064	0.006	50.002%	102,302
200,000	20	18.266	0.023	0.000%	130,583
200,000	16	18.493	0.026	0.002%	106,634
200,000	12	18.343	0.023	0.026%	80,762
200,000	8	18.339	0.025	0.384%	61,502
200,000	7	18.444	0.025	0.764%	56,966
200,000	6	18.557	0.024	1.574%	53,381
200,000	5	18.306	0.024	3.146%	50,663
200,000	4	18.091	0.021	6.268%	49,724
200,000	3	18.672	0.022	12.542%	51,962
200,000	2	18.573	0.020	25.034%	62,339
200,000	1	18.390	0.013	49.922%	105,230
500,000	20	47.358	0.066	0.000%	315,083
500,000	16	46.731	0.061	0.000%	254,234

500,000	12	45.979	0.058	0.025%	186,194
500,000	8	46.450	0.065	0.385%	132,920
500,000	7	46.532	0.059	0.775%	118,466
500,000	6	46.441	0.060	1.550%	106,097
500,000	5	46.526	0.058	3.115%	94,943
500,000	4	45.651	0.052	6.255%	84,866
500,000	3	47.220	0.054	12.580%	78,320
500,000	2	47.097	0.047	25.000%	79,913
500,000	1	46.266	0.032	49.990%	114,014

Chapter 6

Discussion and Future Work

In this chapter we discuss the results from Chapters 4 and 5 and identify several directions for future work.

6.1 Password hash function

In Section 4.4.2, the requirement of a cryptographic and memory-hard hash function for passwords is discussed. Since a user has only one secret (the password), it is not possible to completely prevent bruteforcing a password. However, it can be made harder and thus more expensive by using a slow and memory-hard hash function.

Unfortunately, such a hash function does not exist yet in a SNARK-friendly manner. Since computations in zk-SNARKs are significantly slower than when performed outside a zk-SNARK, existing hash functions suitable for passwords cannot be duplicated for use in a zk-SNARK, because they would be either too expensive to use or too fast outside a zk-SNARK, defeating its security.

While using a more efficient hash function eases bruteforcing, it is only the server that can use this to its advantage. After all, only the password hash encryptions are stored in the database, and assuming the SAVER private key is stored securely (e.g. in a Hardware Security Module) and not compromised, adversaries cannot bruteforce the password hash encryptions if the database is leaked. Hence, this requires more trust in the server, but does not impact password confidentiality if the database is compromised.

Future work is oriented at two directions. First, more research in password hashing functions for zk-SNARKs might uncover new hash functions that are SNARK-friendly and suitable for use in zk-SNARKs. Secondly, research can be aimed at modifying or replacing SAVER, such that the individual ciphertexts cannot be decrypted, while still allowing an equality check of the encrypted plaintexts. This eliminates the ability to bruteforce passwords.

6.2 Proving speed

Section 5.2 showed that creating a zk-SNARK registration proof can easily take a few seconds, depending on the used password policies. This is quite long in comparison with the current practise of sending the password in plaintext to the server, which does not incur any additional waiting time. In addition, the zk-SNARK proof is created by the client, and hence the proving speed is heavily influenced by the power of the device. Weaker smartphones, for instance, may have a significantly longer proving time.

However, as a user only registers theirself once, it is a one-time delay and hence may be acceptable. In addition, future work may be aimed at optimizing the implementation, improving the proving time. Verifying the registration and login proofs and creating the login encryption all perform well under a second, and are thus fast enough to be practical.

6.3 Fetching salts

Because the password is hashed on the client, as described in Section 4.4.3, the password hash salt needs to be fetched from the server during login. This salt can for example be stored in a public directory or exposed via an API call. However, this exposes which usernames are taken, since usernames that are not registered do not have a salt. This may be considered a security threat, since this information can be used to bruteforce username and password combinations.

While a public directory cannot hide taken username, it is possible when exposing salts via an API. Instead of returning an error when the salt of a non-registered username is requested, a dummy salt may be returned. Naturally, returning the same dummy salt does not work, as real salts and dummy salts can be easily distinguished. On the other hand, returning a different random salt will also not work, as two invocations of the API reveal that a dummy salt is used. Another solution is to save the random salt for a non-registered username, such that it can be returned upon further invocations. However, this produces a large storage cost, potentially having to store a salt for every possible username.

Instead, a better solution is to use an HMAC for non-registered usernames, with the username as input and a key exclusively known to the server. By using a cryptographic hash function in the HMAC, the result is indistinguishable from random and can thus not be distinguished from a real salt. In addition, the dummy salt for each username is consistent over multiple invocations. Finally, since the key is not known to anyone but the server, the hashes cannot be computed and compared by an adversary.

Chapter 7

Conclusion

In this thesis, we presented zk-SCHNAPS. We showed that arbitrary password policies can be enforced by leveraging a zk-SNARK in the registration phase, proving compliance to the implemented password policies. The zk-SNARK is combined with SAVER to yield an encryption that can be stored by the server for later authentication. During login, a similar encryption is created and the ciphertexts are compared by the server using the homomorphic property of SAVER.

We presented how password policies can be encoded in a zk-SNARK and showed the implementation of the most common ones. In addition, we showed the feasibility of the protocol by providing an implementation, and measured the performance by running several benchmarks. We found that creating the zk-SNARK proof in the registration phase takes a few seconds depending on the implemented password policies, with all other protocol function taking well under one second, showing practical performance.

Bibliography

[ABLZ17] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michał Zając. A Subversion-Resistant SNARK. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10626 LNCS:3–33, 2017.

[AGR$^+$16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10031 LNCS, pages 191–219, 2016.

[AHH22] Michel Abdalla, Björn Haase, and Julia Hesse. CPace, a balanced composable PAKE. Internet-Draft draft-irtf-cfrg-cpace-05, Internet Engineering Task Force, jan 2022.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2087–2104, 2017.

[AW21] Thomas Attema and Daniel Worm. Eindelijk een privacyvriendelijke manier om data te benutten. Technical report, TNO, 2021.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *ITCS 2012 - Innovations in Theoretical Computer Science Conference*, pages 326–349, 2012.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 292–302. IEEE, 2016.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive Proof Composition without a Trusted Setup. *IACR Cryptology ePrint Archive*, (2019/1021):1–31, 2019.

[BHW20] Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification. page 156, 2020.

[Blo70] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM92] Steven M Bellovin and Michael Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *Proceedings of the Symposium on Security and Privacy*, pages 72–84, 1992.

[BM93] Steven M Bellovin and Michael Merritt. Augmented Encrypted Key Exchange: a Password-Based Protocol Secure Against Dictionary Attacks and Password File Compromise. In *1st ACM Conference on Computer and Communications Security*, pages 244–250, 1993.

[BmIMt$^+$22] Marta Bellés-muñoz, Miguel Isabel, Jose Luis Muñoz-tapia, Albert Rubio, and Jordi Baylina. CIRCOM: A Robust and Scalable Language for Building Complex Zero-Knowledge Circuits. pages 1–16, mar 2022.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Eprint.Iacr.Org*, (693423):1–83, 2018.

[BSCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[BSCR$^+$19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent Succinct Arguments for R1CS. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11476 LNCS, pages 103–128, 2019.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *Proceedings of the 23rd USENIX Security Symposium*, pages 781–796, 2014.

[CFF+21] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: A Toolbox for More Efficient Universal and Updatable zkSNARKs and Commit-and-Prove Extensions. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 13092 LNCS, pages 3–33, 2021.

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile Verifiable Computation. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2015-July, pages 253–270, 2015.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12105 LNCS, pages 738–768, 2020.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-Quantum and Transparent Recursive Proofs from Holography. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12105 LNCS, pages 769–793, 2020.

[DMR10] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Measuring Password Strength: An Empirical Analysis. In *Proceedings - IEEE INFOCOM*, jul 2010.

[Erk15] Z. Erkin. Private Data Aggregation with Groups for Smart Grids in a Dynamic Setting using CRT. In *2015 IEEE International Workshop on Information Forensics and Security, WIFS 2015 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., dec 2015.

[ET12] Zekeriya Erkin and Gene Tsudik. Private Computation of Spatial and Temporal Power Consumption with Smart Meters. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7341 LNCS, pages 561–577. Springer, Berlin, Heidelberg, 2012.

[FAKM14] Bin Fan, David G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*

2014 - Proceedings of the 2014 Conference on Emerging Networking Experiments and Technologies, pages 75–87, 2014.

[FH07] Dinei Florencio and Cormac Herley. A Large-Scale Study of Web Password Habits. In *16th International World Wide Web Conference, WWW2007*, pages 657–666, 2007.

[FH10] Dinei Florêncio and Cormac Herley. Where Do Security Policies Come From? In *ACM International Conference Proceeding Series*, 2010.

[FS87] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 263 LNCS, pages 186–194. Springer Verlag, 1987.

[GGF17] Paul A Grassi, Michael E Garcia, and James L Fenton. Digital Identity Guidelines. Technical Report 63, 2017.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7881 LNCS, pages 626–645, 2013.

[GKR$^+$21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems. In *Proceedings of the 30th USENIX Security Symposium*, pages 519–535, 2021.

[GL20] Thomas Mueller Graf and Daniel Lemire. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM Journal of Experimental Algorithmics*, 25, dec 2020.

[GM17] Jens Groth and Mary Maller. Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10402 LNCS, pages 581–612. Springer Verlag, 2017.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof-Systems. *SIAM Journal on Computing*, 18(1):186–208, jul 1989.

[Gro16] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9666, pages 305–326. Springer, Berlin, Heidelberg, 2016.

[GWC20] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. PLONK : Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *Stanford Blockchain Conference*, pages 1–33, 2020.

[HL19] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):1–48, feb 2019.

[HR10] Feng Hao and Peter Ryan. J-PAKE: Authenticated Key Exchange without PKI. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6480(PART 2):192–206, 2010.

[ide] iden3. Github - iden3/snarkjs: zkSNARK implementation in JavaScript & WASM. `https://github.com/iden3/snarkjs`.

[Jab96] D. P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, oct 1996.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10822 LNCS, pages 456–486, 2018.

[KLO20] Jihye Kim, Jiwon Lee, and Hyunok Oh. Simulation-Extractable zk-SNARK with a Single Verification. *IEEE Access*, 8:156569–156581, 2020.

[KM14] Franziskus Kiefer and Mark Manulis. Zero-Knowledge Password Policy Checks and Verifier-Based PAKE. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8713 LNCS, pages 295–312, 2014.

[KM16] Franziskus Kiefer and Mark Manulis. Blind Password Registration for Verifier-based PAKE. In *AsiaPKC 2016 - Proceedings of the 3rd ACM*

International Workshop on ASIA Public-Key Cryptography, Co-located with Asia CCS 2016, pages 39–48, 2016.

[KPPS20] Ahmed Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal zk-SNARKs. In *Proceedings of the 29th USENIX Security Symposium*, pages 2129–2146, 2020.

[KSK$^+$11] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of Passwords and People: Measuring the Effect of Password-Composition Policies. In *Conference on Human Factors in Computing Systems - Proceedings*, pages 2595–2604, 2011.

[LCKO19] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. SAVER: SNARK-friendly, Additively-homomorphic, and Verifiable Encryption and decryption with Rerandomization. *Cryptology ePrint Archive*, pages 1–37, 2019.

[Mie19] Daniel Miessler. GitHub - SecLists/xato-net-10-million-passwords.txt at master · danielmiessler/SecLists. `https://github.com/danielmiessler/SecLists/blob/master/Passwords/xato-net-10-million-passwords.txt`, 2019.

[MKBM19] Mary Maller, Markulf Kohlweiss, Sean Bowe, and Sarah Meiklejohn. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2111–2128, 2019.

[Ng19] Alfred Ng. Google had some passwords stored in plaintext for more than a decade - CNET. `https://www.cnet.com/news/privacy/google-had-some-passwords-stored-in-plaintext-for-more-than-a-decade/`, 2019.

[Ped92] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 576 LNCS, pages 129–140. Springer Verlag, 1992.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2161:121–133, 2001.

[PS96] David Pointcheval and Jacques Stern. Security Proofs for Signature Schemes. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1070, pages 387–398. Springer Verlag, 1996.

[SAD20] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-Prime: a Standard Specification (SoK). pages 1–16, 2020.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 12172 LNCS, pages 704–737, 2020.

[Sho97] Victor Shoup. Lower Bounds for Discrete Logarithms and Related Problems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1233, pages 256–266. Springer Verlag, 1997.

[SKD$^+$16] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Designing Password Policies for Strength and Usability. *ACM Transactions on Information and System Security*, 18(4):13, 2016.

[Smy20] Stanislav V. Smyshlyaev. [Cfrg] Results of the PAKE selection process. `https://mailarchive.ietf.org/arch/msg/cfrg/LKbwodpa5yXo6VuNDU66vt_Aca8/`, 2020.

[WTS$^+$18] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2018-May, pages 926–943, 2018.

[Wu98] Thomas Wu. The Secure Remote Password Protocol. *Proceedings of the Symposium on Network and Distributed Systems Security NDSS 98*, 4:97–111, 1998.

[XZZ$^+$19] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11694 LNCS, pages 733–764, 2019.

[Zer19] Zerocoin Electric Coin Company (ZECC). Privacy-protecting digital currency — Zcash. `https://z.cash/`, 2019.