Lise Millerjord

Picnic with Friends:

Constructing post quantum digital signature schemes

Master's thesis in Mathematical Sciences
Supervisor: Kristian Gjøsteen
June 2021

Lise Millerjord

Picnic with Friends:

Constructing post quantum digital signature schemes

Master’s thesis in Mathematical Sciences
Supervisor: Kristian Gjøsteen
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Summary

The PICNIC [5] digital signature scheme is a post-quantum scheme introduced as part of the NIST standardisation process for post-quantum cryptography [15]. We look at how PICNIC is constructed and its security properties. We begin by introducing several cryptographic concepts that are used in the scheme, among these proofs of knowledge, zero knowledge, one-way functions and multi-party computation.

PICNIC consists of a one-way function, which is instantiated with the block cipher LowMC [1], and a non-interactive zero knowledge (NIZK) proof of knowledge, which is instantiated with ZKB++. ZKB++ is constructed based on the sigma protocol ZKBoo [10], which is a zero knowledge proof of knowledge. ZKBoo also uses a technique called MPC-in-the-head to achieve honest verifier zero knowledge (HVZK).

Some optimisations are made to ZKBoo, and the protocol is made non-interactive using Unruh's transform [18], resulting in the ZKB++ scheme. The resulting signature scheme, PICNIC, has EUF-CMA security, which reduces solely to symmetric-key primitives.

Sammendrag

Det digitale signatursystemet, PICNIC [5], er et post-kvantesystem som ble introdusert som en del av NIST-prosessen for å standardisere post-kvantekryptografi [15]. Vi ser på hvordan PICNIC er konstruert, og sikkerhetsegenskapene til systemet. Vi begynner med å introdusere noen kryptografiske konsepter som er brukt i systemet, blant annet kunnskapsbeviser, kunnskapsløse bevis, enveisfunksjoner og flerpartsberegning.

PICNIC består av en enveisfunksjon, som er instansiert med blokkchifferet LowMC [1], og et NIZK-kunnskapsbevis, det vil si et ikke-interaktivt kunnskapsbevis som ikke avslører hemmeligheten ("kunnskapsløst"), som er instansiert med ZKB++. ZKB++ er konstruert fra sigma-protokollen ZKBoo [10], som er et kunnskapsbevis som ikke avslører hemmeligheten. ZKBoo bruker også en teknikk som kalles MPC-in-the-head for å oppnå kunnskapsløshet med en ærlig verifisierings-part (HVZK).

Noen optimaliseringer er gjort for å forbedre ZKBoo, i tillegg til at protokollen er gjort ikke-interaktiv ved å bruke Unruh's transform [18]. Dette resulterer i ZKB++-protokollen. Signatursystemet vi får fra dette, PICNIC, har sikkerhet mot EUF-CMA-angrep og reduserer bare til symmetrisk-nøkkel-primitiver.

Preface

This thesis is submitted as a part of the completion of my master's degree, MSc in Mathematical Sciences, at the Department of Mathematical Sciences, NTNU. The work was supervised by Professor Kristian Gjøsteen and I would like to thank my supervisor for all the knowledge he has shared with me and our many enjoyable and helpful conversations.

Throughout the process of writing the thesis and completing the degree, I have enjoyed the company of all the members of the NACL research group at both the Department of Mathematical Sciences and the Department of Information Security and Communication Technology. My friends and colleagues at these two departments have really made a difference these past years and I expect they will continue to do so in the next phase of my academic career.

Finally, I would like to thank my family and close friends for always believing in me and supporting me.

This work does not contribute to the state of the art, but rather aims to capture the ideas behind the rather complicated signature scheme that is its main topic and explain how it works. There are many different concepts which make up the whole, and the way that the individual parts work, their purpose, as well as how they are put together is what I aim to convey.

Lise Millerjord
Trondheim, June 2021

A river cuts through rock,
not because of its power,
but because of its persistence.

Jim Watkins

Table of Contents

Chapter 1

Introduction

When we are sending messages to our correspondents over the internet, there are several things we would like to have: often we want the content of the message to remain secret (*confidentiality*), we want to be confident that the person who sent the message is actually the person they claim to be (*authentication*), and we want to be sure that the message has not been changed by some third party (*integrity*).

We do not necessarily want all of these properties for all messages that we are sending and receiving. In this thesis we will focus on the authentication and integrity properties. In traditional letter correspondence we achieve this by *signing* our letter before sending it. If the recipient has seen my signature before, they can be reasonably sure that I was in fact the person who sent this letter.

In electronic communication, we want a similar notion to the traditional signature. This is what we call a digital signature. In this thesis, we will introduce in detail a particular construction that achieves this, a digital signature scheme called PICNIC, and we will show that it is *secure*.

For the past several decades, several such digital signature schemes have been designed and used widely. The schemes are either symmetric-key or public-key schemes, but we will be focusing on the public-key digital signature schemes. These schemes typically rely on number-theoretic hardness assumptions. As long as we are all using classical computers, this is not a problem. However, with the emerging threat of quantum computers, we need to design new schemes that are not vulnerable to quantum attacks. In particular, Shor's algorithm [17] shows that two very common classical hardness assumptions, factorisation and discrete logarithms, are not hard with quantum computers.

This is the background for the study of post-quantum cryptography, that is, protocols that are quantum resistant. In 2016 NIST (the National Institute of Standards and Technology, U.S. Department of Commerce) initiated a process to standardise such post-quantum public key protocols [6]. Submissions were made in one of two categories: Public key encryption and key establishment, and digital signatures. The standardisation process has had several rounds of elimination and in June 2020 the round 3 finalists were announced.

PICNIC is a class of quantum-resistant digital signature schemes that was submitted as part of this process. It is not one of the finalists, but is an alternate digital signature scheme that may be considered for standardisation at a later point after further consideration [15]. We will look at how PICNIC is constructed in detail and what hardness assumptions are made when proving its security.

1.1 A general approach for designing signature systems

We would like to come up with a design for a signature system that will be secure against attackers with access to quantum computers. We will begin by looking at some established designs, see why they do not work and attempt to make changes to achieve our goal. Our first attempts do not turn out to work, but we end up with the design that PICNIC is based on.

A very simple example of a signature scheme is based on RSA:

Example 1. (RSA signatures)

Let m be a message we want to sign, and (d, e) be the RSA key pair.

Then we have that the signature, σ is:

$$\sigma \leftarrow m^d = f(m),$$

where $f(\cdot)$ is a secret function, that is the secret key d is not shared.

The verifier checks the signature by checking that:

$$\sigma^e = f^{-1}(\sigma) \overset{?}{=} m,$$

where e is the public key, so $f^{-1}(\cdot)$ is public.

If the RSA signature is verified, then the recipient can be confident that the message was not changed and that only someone who knows the secret key could have signed the message. The private key and the public key are mathematically linked in such a way that only the private key could have signed a message that is verified with the public key. However, this is based on the RSA cryptosystem, which relies on the hardness of factoring. Therefore, this is not quantum-resistant.

Another approach to create a signature scheme is to let the signature consist of both some secret function evaluated in the message and a proof that we evaluated the function correctly. This proof would need to be non-interactive since we do not want the signature scheme to require several messages and in addition, we need the zero knowledge property, that the proof doesn't leak any information about the secret key.

Example 2. (Evaluate-then-prove signature)

Let $G = \langle g \rangle$ be a group with generator g. Let x be the secret key and $y = g^x$ be the public key.

Then the signature of message m will be the :

$$\sigma = (s, \pi) \leftarrow (m^x,\ \mathsf{NIZK}(log_g y = log_m s)),$$

where $\mathsf{NIZK}(log_g y = log_m s)$ is a non-interactive zero knowledge proof for that the state-

ment $log_g y = log_s$ is true.

So the secret function we evaluate is $f(m) = m^x$, and then we prove that we evaluated the function correctly by showing that the signature and the public key are images of the same function.

Example 2 describes how the Scnorr signature scheme works (Example 7 in Section 2.2). Similar to the RSA signatures, there is a unique secret key that was used to sign the message, and if the NIZK proof is sound, then the recipient can authenticate the sender. However, like with the RSA signatures, this relies on a number theoretic hardness assumption that doesn't hold against quantum adversaries, the discrete logarithm problem in this case.

So the problem with Example 2 is that the secret function is no longer secret in the context of quantum computing. However, we can generalise this idea and consider any secret function with desirable properties and then a NIZK proof that the function was evaluated correctly:

Example 3. (General evaluate-then-prove)

*Let $f_k : A \to B$ be some secret function with domain A and range B. We assume that the function is "*secure*", by which we mean that given (a, b) such that $b = f_k(a)$, it is hard to find the correcy key k. That is, the function remains secret even if we know some function values.*

Let the secret key be $sk = k$ and the public key $pk = (a, b)$. Now the signature will consist of an evaluation of this function in the message m as well as a NIZK *proof that the function was evaluated correctly:*

$$\sigma = (s, \pi) \leftarrow (f_k(m), \mathsf{NIZK}(\exists k : f_k(a) = b \textit{ and } f_k(m) = \sigma)).$$

This seems like a good strategy for creating a signature scheme. However, in practice we run into problems here too. This is because we may not in practice be able to say that there will be only one key k that satisfies this condition. In fact, it may be relatively easy to find *some* key that satisfies the condition, but that is not the actual key that was used. This is because anyone wanting to prove this only needs to find a key that yields a function that agrees in two points. So we would need to require from our function f_k that it is hard to find such a key, and this may not be a reasonable assumption for functions we may wish to use.

To avoid this issue of an adversary just needing to find one of many possible keys in order to forge a signature, we want to make the adversary have to find the specific key that was used. So now we want the signature to include a NIZK proof of knowledge of the secret key. Simply adapting the scheme from Example 3, the signature would look like this:

$$\sigma = (s, \pi) \leftarrow f_k(m), \mathsf{NIZK}(\text{"I know } k \text{ such that } b = f_k(a)\text{"}).$$

Proving that you have access to the secret key k authenticates the sender, but this is no longer a signature, since it does not show that the key k is used in the evaluation of m. In fact, the NIZK proof is not tied to the message in any way, and thus anyone with access to this message-signature pair can create a new valid pair by choosing a new message, evaluating some function in the message and attaching the same NIZK proof. This can be fixed by including in the NIZK proof that "I know the key k such that $b = f_k(a)$ *and* it is the same as the key k' s.t $s = f_{k'}(m)$". This can get very complicated so we would prefer not to do this.

Instead of doing this, we would rather make the NIZK proof of knowledge depend on the message m in some way. This would mean that the proof will only verify as correct when it is verified with the specific message. In practice, we are going to create these proofs using a sigma protocol that is a zero knowledge proof of knowledge and then applying a non-interactive transformation to it, the Fiat-Shamir transform or Unruh's transform. Both of these allow us to embed the message in the challenge in a way such that the proof will only be valid if the correct message is used in the verification.

Now that we are embedding the message in the proof, there is no longer any need to include an evaluation of the secret function in the message, because that does not provide any useful information or fulfil any security purpose. So the signature will just consist of a NIZK proof of knowledge of the secret key, that depends on the message. Our new signature system looks like this:

Example 4. (A general signature scheme)
The KeyGen *algorithm chooses a key k for a one-way function f_k and a secret key a, computes $b \leftarrow f_k(a)$, and outputs the public key $pk = (k, b)$ and the secret key $sk = (a, pk)$.*

The Sign *algorithm takes a message m and the secret key sk and outputs a signature: $\sigma = (s, \pi)$,*

$$\sigma = \mathsf{NIZK}(\text{"}\textit{I know } f_k^{-1}(b)\text{"}),$$

where the challenge in the proof is computed with the message.

The Verify *algorithm takes a message m, a signature σ and the public key pk and verifies the* NIZK *proof of knowledge. It outputs* Accept *if and only if this verification succeeds for message m.*

Note that since $f_k(\cdot)$ is a one-way function, the preimage a (which is the secret key) cannot be found given the image b and the key k. Since the NIZK proof is a proof of knowledge, anyone who does not have access to the secret key cannot produce a forgery that is a valid proof. This is because there exists an extractor that finds the witness (the secret) from valid proofs, so if an adversary can reliably produce a valid forgery, then they also have access to the secret. Lastly, since the NIZK proof is zero knowledge, then no information about the secret key leaks from the (arbitrary number of) proofs.

This signature construction is how the PICNIC signature scheme works. In the following chapters we will describe both what one-way function PICNIC uses and how the NIZK proof of knowledge is constructed.

1.1.1 What do we bring to the PICNIC?

The PICNIC signature scheme is an example of the construction in Example 4.

The one-way function in the construction is in PICNIC considered to be a block cipher (in particular, the block cipher LowMC [1]). So b is the encryption of a under key k. Note that for a block cipher, it does not make sense to give away the key and the image, and then assume it is hard to find a preimage. Given the key, it is trivial to find a preimage by just decrypting. Because of this, we define the one-way function to be the mapping of the set of block cipher keys $k \in K$ to the set of images $b \in R$ in the range of the block cipher, given a (fixed) preimage a:

$$\begin{aligned} f_a : K &\to R, \\ k &\mapsto b, \text{ such that } b = \mathsf{Enc}_k(a). \end{aligned}$$

This is a reasonable construction because given a plaintext-ciphertext pair (a, b) such that $b = \mathsf{Enc}_k(a)$, it should be hard to find the key k, otherwise the block cipher is not secure (the block cipher is not a secure pseudorandom permutation).

Furthermore, we need a NIZK proof of knowledge for the block cipher key k. The construction that PICNIC uses for this is based on a zero knowledge sigma protocol called ZKBoo [10]. Some alterations are made to this protocol in order to increase efficiency. Additionally, Unruh's transform is employed to make the resulting protocol non-interactive. This non-interactive zero knowledge proof of knowledge protocol is called ZKB++ [5].

The original sigma protocol ZKBoo is a zero knowledge proof of knowledge for a preimage a of a function ϕ such that the public value $b = \phi(a)$. This is achieved by using a technique called MPC-in-the-head [12]. This technique uses the link between zero knowledge and secure multiparty computation: if an MPC protocol is secure that means that no information apart from the output value is leaked. This is exactly what we want to achieve when proving knowledge of some input (preimage) to a function: we want to prove that the output of the function is what we claim, but without revealing the input.

We represent the function that we want to evaluate as an algebraic circuit and then we decompose this circuit in a specific way: the decomposed circuit will be evaluated by separate players on their own private input such that the input for all players put together is the input of our original function and the output of all the players put together is the output of the original function. Specifically, there are three players and we have the property that revealing the view of any two players will not reveal any information about the input, which is our secret key. However, without knowing the input, the person simulating the three players will not be able to create a correct and consistent view for all three players, and since the verifier chooses the challenge, that is which players views to open, the prover is unable to reliably succeed in cheating.

This is still an interactive protocol, and in order to get to the non-interactive protocol we want, we employ Unruh's transform [18]. This transformation describes how to create accepting conversations and compute challenges in such a way that it preserves the security properties of the original protocol.

The details of the constructions and concepts are included in the following chapters.

1.1.2 Friends of PICNIC

Since the introduction of the PICNIC signature scheme, other signatures schemes with a similar construction have been presented. Two of these schemes are BBQ and BANQUET.

The high level constructions of all three of these signature schemes are the same, and they differ mostly by what building blocks they have chosen to instantiate with.

The BBQ signature scheme [8] uses the same NIZK proof of knowledge as PICNIC (and BANQUET), and its main difference is the use of the standard block cipher AES instead of LowMC. Since AES has been studied significantly more than LowMC, the confidence in the security of AES is higher than for LowMC. However, this increased confidence in the security of the scheme comes at the cost of larger sigantures. The PICNIC signatures are already quite large, thus any increase in signature sizes from using AES the way that BBQ does, is unwanted. However, it is an important contribution to see how the scheme performs when using standard primitives, and the next scheme builds on this work.

The BANQUET signature scheme [2] also uses the standard block cipher AES instead of LowMC. However, they introduce a new technique for evaluating the AES circuit such that the signature size is significantly reduced compared to BBQ, and competitive with the PICNIC scheme. This comes at the cost of longer run time, where it takes longer to sign messages than it does using PICNIC. Compared to BBQ, BANQUET represents a significant improvement in performance. Since BANQUET uses only standard primitives and is competitive with PICNIC in terms of signature sizes, this shows that the general signature construction has potential to produce practical and very secure signature schemes.

We will not be focusing on other schemes than PICNIC going forward, but we note that the security results for PICNIC hold for any signature scheme with this construction, and only depend on the security of the building blocks. So the work is relevant not only for the PICNIC scheme specifically, but for any future signature scheme like this, that may have even more desirable properties in terms of signature sizes and signing times.

1.2 Structure of thesis

We begin by introducing some relevant theory in Chapter 2. We expect the reader to be familiar with the basics of cryptography and algebra, but we will include the concepts that are important to understand the following chapters. This introduction is followed by an introduction to proofs of knowledge and zero knowledge in Chapter 3. We also include some security definitions and show how we can make the protocols we are interested in non-interactive. This will be important as the PICNIC signature construction relies on a non-interactive zero knowledge (NIZK) proof of knowledge.

We then move on to introduce block ciphers in Chapter 4. We explain how we can construct a one-way function from a block cipher and discuss some properties of block ciphers that are of interest to us. We also introduce two specific block ciphers, one of which is used in the PICNIC construction.

The last significant concept we need to construct our signature scheme is multiparty computation. This is introduced in Chapter 5 along with a specific technique called MPC-in-the-head that we will use in our proof of knowledge.

After having seen all the relevant background, we introduce the proof of knowledge protocol that we will be using in Chapter 6. The protocol used in PICNIC is called ZKB++, and is based on another protocol called ZKBoo and it is made non-interactive by applying Unruh's transform.

Finally, we describe the PICNIC signature scheme in Chapter 7, and prove that it is secure.

Chapter 2

Background

In this chapter we will introduce some theory that we assume the reader is familiar with, but we include it in order to introduce notation and clarify what definitions we are using in the remaining chapters and so that this document is more self-contained.

In addition to the material cited, the theory in this and the following chapters comes from standard textbooks, including [13, 11, 4, 7, 14].

2.1 Basic definitions

We informally introduce some elementary definitions that are useful when discussing cryptography in general.

In a cryptographic protocol we usually talk about *parties* that execute a protocol. For example, in a digital signature scheme, we have two parties: a signer and a verifier. However, when we talk about the security of such a protocol, we do that in terms of adversaries. An *adversary*, or attacker, is an entity that is dishonest in some way, either by trying to learn information they should not have access to or by doing things that they are not supposed to be doing.

In the case of a digital signature scheme, the adversary may try to produce forgeries of signatures in order to fraudulently convince a verifier, or they may be interacting with a signer to get signatures, in order to learn some secret information from these signatures. We sometimes also consider a third person to be an adversary. This is an entity that is not participating in the execution of the protocol, but who is listening to the conversation in order to try to learn something.

When we define security, we typically design an *experiment*, or a game, which the adversary plays against a *challenger*. If the adversary wins the game, it has succeeded in breaking the security that the game specifies for the cryptographic protocol. We will define the security of a system against an attack in terms of the *advantage* the adversary in the security experiment. The advantage tells us how likely it is that this adversary can win the game in a meaningful way. We have to make this distinction because of the different nature of different security experiments.

Sometimes, the adversary needs to guess one out of two possible values in order to win, and it will have some probability of success when doing this. In this case, anyone who makes a random guess will win the game with probability ½. So we are actually interested in how much better the adversary is at guessing compared to someone making random guesses. In cases like this we define the advantage of our adversary $\mathcal{A}$ to be

$$Adv(\mathcal{A}) = |Pr(\mathcal{A}\text{ succeeds}) - \frac{1}{2}|.$$

Note the use of absolute values. This is because if we have an adversary $\mathcal{B}$ that has success probability significantly less than ½, then we can use $\mathcal{B}$ to create an adversary $\mathcal{A}$ that has success probability significantly higher than ½ by just having $\mathcal{A}$ make the opposite choice to $\mathcal{B}$.

In other security experiments, the adversary just needs to win once in order for the security of the system to be compromised. In this case we define the advantage of the adversary $\mathcal{A}$ to be

$$Adv(\mathcal{A}) = Pr(\mathcal{A}\text{ succeeds}).$$

We do not require *perfect*, or information-theoretic, security from cryptographic schemes, as that is usually very impractical. Instead, we consider computational security, where we consider a cryptographic scheme to be secure if it leaks only a ”small”, amount of data to an attacker that has bounded computational power. So we are making two concessions: The advantage of the adversary does not need to be zero, and we allow ourselves to assume how powerful our adversary is.

When we discuss the advantage of the adversary, we consider a scheme to be secure if the advantage is *negligible*. In other words, it is so small, that we can safely disregard it. We also talk about negligible functions, which intuitively have the same meaning: The function values are so small compared to the input values, that we think of it as almost zero.

In regard to the power and abilities of the adversary: We consider an algorithm to be *efficient* if it has limited computational power and it terminates ”fast enough”. This is justified since any real world machine running an algorithm will in fact have finite computing power and cannot spend an infinite amount of time. In a similar manner, we classify a problem as *hard* if there is no efficient algorithm that reliably solves it.

Sometimes we give an algorithm access to what we call *oracles*. An oracle is just another algorithm that has access to some information and that our algorithm can call upon with input of its own choosing. We distinguish between two different types of oracles.

Random oracles are idealised functionality that replace some actual evaluation done by each party in a protocol. It can for example replace a cryptographic hash function in security analysis. These oracles do not actually exist, and we are not claiming that they do, we are just simplifying our security analysis by assuming some strong randomness properties.

The other kind of oracle is something that can actually exist and that typically has access to some secret information. For example, we can give an adversary access to a signing oracle. This means that the adversary can query the signing oracle with messages and receive signatures for the messages in return. The oracle has access to the secret key used for signing, which the adversary does not have. So the oracle doesn't do anything that the original signer cannot do, we are simply using the oracle to determine how the adversary has access to requesting signatures on messages.

There are many more concepts and ideas that could be included, and the concepts included here are formally defined in the literature. However, for the purposes of understanding the content of the remaining chapters, this formality is not necessary. Thus we leave it at these vague, intuitive ideas.

2.2 Digital Signatures

A digital signature scheme is a cryptographic scheme that lets you *sign* messages. This is comparable to a normal signature with pen on paper. When you physically sign a document, you are saying that you endorse the content of the document, and the signatures uniqueness ties it to your identity. So the digital signature can provide integrity and authenticity in messages. The signature must be verifiable and hard to forge for it to be trustworthy. When a secure digital signature scheme is used, a valid digital signature will convince the receiver that the message was signed by the correct person and that the content of the message has not been altered since it was signed. Digital signature schemes are formally defined in Definition 1. Figure 2.1 visualises how the algorithms interact.

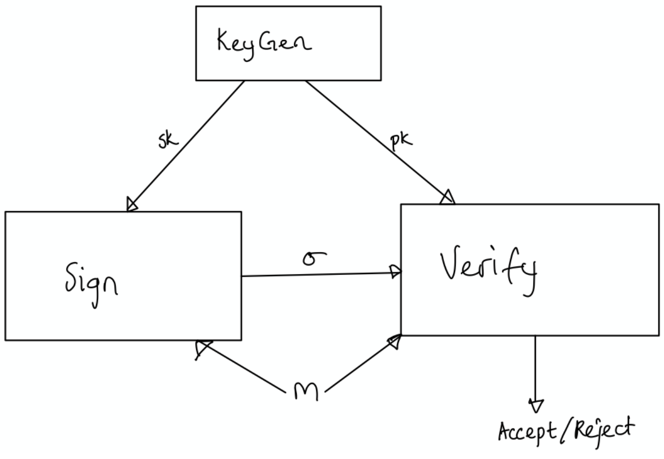


Figure 2.1: Illustration of a signature scheme.

Definition 1. (Digital signature scheme)
*A digital signature scheme is a triple of efficient algorithms (*KeyGen, Sign, Verify*) such that:*

- *The key generation algorithm,* KeyGen, *outputs the private and public keys:* $(pk, sk) \leftarrow \mathsf{KeyGen}()$.
- *The signing algorithm,* Sign, *takes a message and the private key and outputs a*

signature: $\sigma \leftarrow \mathsf{Sign}(m, sk)$.

- *The verification algorithm,* Verify*, takes the public key, a message and a signature, and outputs* Accept *or* Reject*:* $b \leftarrow \mathsf{Verify}(pk, m, \sigma)$*, where* $b \in \{\mathsf{Accept}, \mathsf{Reject}\}$.
- *(*KeyGen, Sign, Verify*) is* correct*, that is* $(pk, sk) \leftarrow \mathsf{KeyGen}() \implies \mathsf{Accept} \leftarrow \mathsf{Verify}(pk, m, \mathsf{Sign}(m, sk))$.

Note that we require *correctness* from our digital signature scheme. So if we have a tuple $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, but this scheme does not have correctness, it is in fact not a digital signature scheme. The justification for this requirement is that if the scheme does not have correctness, then correctly generated signatures can fail to verify, which means the scheme is unreliable and thus no longer useful.

Example 5. (Schnorr signatures)
The Schnorr signature scheme is based on the discrete logarithm problem. This scheme is an example of the signature scheme presented in Example 2. The parties agree on a group G with generator g, in which we assume the discrete logarithm problem is hard. They also agree on a cryptographic hash function H.

Key generation: $(pk, sk) \leftarrow \mathsf{KeyGen}()$
Choose a private signing key, x. The public key is $y = g^x$.

Signing: $(s, e) \leftarrow \mathsf{Sign}(m, x)$. *Let m be the message we want to sign. To sign the message:*

1. *Choose random k,*
2. *Compute $e \leftarrow \mathsf{H}(r \parallel m)$,*
3. *Compute $s \leftarrow k - xe$.*

The sender sends the signature $\sigma = (s, e)$ to the verifier.

Verifying: $b \leftarrow \mathsf{Verify}(pk, m, \sigma)$. *To verify the signature:*

1. *Compute $r_V \leftarrow g^s y^e$,*
2. *Compute $e_V \leftarrow \mathsf{H}(r_V \parallel m)$,*
3. *If $e_V = e$, output* Accept*. Otherwise output* Reject.

Correctness: *We have $r_V = g^s y^e = g^{k-xe} g^{xe} = g^k = r$, and thus we have $e_V = \mathsf{H}(r_V \parallel m) = \mathsf{H}(r \parallel m) = e$. So (*KeyGen, Sign, Verify*) is correct.*

2.2.1 Security of signature schemes

We consider a signature scheme *secure* if it is hard for someone without access to the private key to create signatures that verify as correct. The security requirement is called existential unforgeability under chosen-message attacks (EUF-CMA), and this is the standard security definition for signature schemes. It captures the ability of any adversary to forge a signature on a message for which they have not already seen a signature under the same key.

We define an experiment for an adversary $\mathcal{A}$ forging signatures:

The $G_{\Pi}^{\textsf{EUF-CMA}}(\mathcal{A})$ experiment proceeds as follows:

1. $(pk, sk) \leftarrow \textsf{KeyGen}()$.
2. $\mathcal{A}$ gets pk and access to a signing oracle $\textsf{Sign}_{sk}(\cdot)$. $\mathcal{A}$ makes signing queries. When $\mathcal{A}$ is done, it outputs a message-signature pair: $(m, \sigma) \leftarrow \mathcal{A}(pk, \textsf{Sign}_{sk}(\cdot))$.
3. Once $\mathcal{A}$ has concluded, the experiment outputs 1 if and only if $\textsf{Verify}(m, \sigma) = 1$ and m was not part of an oracle query made by $\mathcal{A}$.

Define the advantage of an adversary $\mathcal{A}$ in the EUF-CMA security experiment $G_{\Pi}^{\textsf{EUF-CMA}}(\mathcal{A})$ as

$$Adv^{\textsf{EUF-CMA}}(\mathcal{A}) = Pr[G_{\Pi}^{\textsf{EUF-CMA}}(\mathcal{A}) = 1].$$

Figure 2.2: EUF-CMA experiment: $G_{\Pi}^{\textsf{EUF-CMA}}(\mathcal{A})$.

If an adversary $\mathcal{A}$ wins this game, then they have successfully forged a signature. So we define the security of the signature system Π in terms of the probability of an adversary succeeding in $G_{\Pi}^{\textsf{EUF-CMA}}$.

Definition 2. (EUF-CMA **security)**
A signature scheme Π = *(*KeyGen, Sign, Verify*) is* EUF-CMA *secure if for any efficient adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ in the* EUF-CMA *security experiment $G_{\Pi}^{\textsf{EUF-CMA}}(\mathcal{A})$ in Figure 2.2 is negligible.*

Definition 2 is the security definition we will use for signature schemes. However, there are other definitions that are used in other circumstances. One of them is a stronger security definition than EUF-CMA. This is called *strong* existential unforgeability under chosen message attacks (SEUF-CMA).

The difference between EUF-CMA and SEUF-CMA is that in the stronger version, we do not require that the message, m, that the adversary, $\mathcal{A}$, succeeds with has not been included in any oracle queries. Instead, we just require that the pair (m, σ) output by $\mathcal{A}$ is not equal to any pairs of queries made by the adversary and their responses from the signing oracle.

So SEUF-CMA also captures the ability of an adversary to come up with another valid signature σ' on a message m that it has seen some signature σ for such that $\sigma' \neq \sigma$, in addition to the ability to forge a signature on any new message $m' \neq m$.

2.3 Commitment Schemes

A commitment scheme is a cryptographic primitive that lets you commit to a chosen value or statement, m, while still keeping it hidden to others. The commitment, $com \leftarrow \mathsf{Commit}(m)$, is *hiding* if an adversary cannot learn the chosen value m from the commitment, and it is *binding* if you are committed to the value m that you chose - you cannot later pretend that you had chosen a different value $m' \neq m$. The commitment can be *opened* by revealing the value m, and then anyone can verify that the commitment com actually belongs to m. Note that there are schemes for which the opening is not actually the value m that was committed to, but some other value. However, for our purposes we will assume that this is not the case.

One way to visualise this is to think of the commitment as a locked box. You put your chosen value in the box and give the box to the receiver. The receiver cannot open the box, because it does not have the key, so the receiver does not know the value you chose (hiding). Also, since the receiver has the box, you cannot change the value inside the box later (binding). You can only reveal the value to the receiver by giving them the key (the opening) at a later time.

Definition 3. (Commitment scheme)
Let m be the value a sender wants to commit to. A commitment scheme is a tuple of efficient algorithms ($\mathsf{Generate}$, Commit*) such that:*

- *The generating algorithm outputs the public parameters:*
  $params \leftarrow \mathsf{Generate}()$
- *The commitment algorithm takes a message and the public parameters, chooses a random r, and outputs a commitment: $com \leftarrow \mathsf{Commit}(m, params, r)$*

The $G^{\mathsf{Binding}}_{\mathsf{Commit}}(\mathcal{A})$ experiment proceeds as follows:

1. Public parameters are generated: $params \leftarrow \mathsf{Generate}()$.
2. The adversary is given the public parameters and returns some output: $(com, m, r, m', r') \leftarrow \mathcal{A}(params)$.
3. The experiment outputs 1 if and only if $m \neq m'$ and $\mathsf{Commit}(m, params, r) = com = \mathsf{Commit}(m', params, r')$.

Define the advantage of an adversary $\mathcal{A}$ in the binding experiment $G^{\mathsf{Binding}}_{\mathsf{Commit}}(\mathcal{A})$ to be

$$Adv^{\mathsf{Binding}}_{\mathsf{Commit}}(\mathcal{A}) = Pr[G^{\mathsf{Binding}}_{\mathsf{Commit}}(\mathcal{A}) = 1].$$

Figure 2.3: Binding experiment: $G^{\mathsf{Binding}}_{\mathsf{Commit}}(\mathcal{A})$.

The sender will send com to the receiver. When the sender wants to open the commitment, they can send (m, r) to the receiver, who can then check that the commitment

The $G^{\text{Hiding}}_{\mathsf{Commit}}(\mathcal{A})$ experiment proceeds as follows:

1. Public parameters are generated: $params \leftarrow \mathsf{Generate}()$.
2. The adversary outputs two messages: $(m_0, m_1) \leftarrow \mathcal{A}(params)$.
3. The challenger uniformly chooses $b \in \{0, 1\}$, and computes the commitment: $com \leftarrow \mathsf{Commit}(m_b, params, r)$.
4. The adversary gets com and guesses b: $b' \leftarrow \mathcal{A}(com, m_0, m_1, params)$.
5. The experiment outputs 1 if and only if $b' = b$.

Define the advantage of an adversary $\mathcal{A}$ in the hiding experiment $G^{\text{Hiding}}_{\mathsf{Commit}}(\mathcal{A})$ to be

$$Adv^{\text{Hiding}}_{\mathsf{Commit}}(\mathcal{A}) = |Pr[G^{\text{Hiding}}_{\mathsf{Commit}}(\mathcal{A}) = 1] - \frac{1}{2}|.$$

Figure 2.4: Hiding experiment: $G^{\text{Hiding}}_{\mathsf{Commit}}(\mathcal{A})$.

is correct by recomputing it: $com \overset{?}{=} \mathsf{Commit}(m, params, r)$. The security of a commitment scheme is defined in terms of the properties hiding and binding. We design security experiments for these properties as shown in Figure 2.3 and Figure 2.4.

We say that a commitment scheme is secure if the advantage of any adversary in both the hiding experiment and the binding experiment is negligible. We also include a simple example of a commitment scheme, which is also the one that is used the protocol ZKBoo introduced in Chapter 6.

Example 6. (SHA-256 **commitment scheme)**
The cryptographic hash function SHA-256 *(m, r) is a good commitment scheme for value m with randomness r chosen by the sender.*
We consider SHA-256 *to be collision resistant, and thus the commitment is binding. Since* $\mathsf{SHA\text{-}256}(\cdot, r)$ *is a sufficiently good PRF, it is also hiding.*

2.4 Security models

When proving security in cryptography, we have to make some assumptions. This is formalised in the security model we choose to use. In Section 2.1 we introduced the idea that we consider an adversary to be limited in the amount of time it uses and the amount of computational power it has. If this is the only assumption we make, then we are working in the *standard model*.

If our cryptographic scheme relies on some computational assumption, stating that this problem cannot be solved by an efficient algorithm, then we can prove the scheme to be secure in the standard model.

However, in many cases we need to make other assumptions. In particular, when schemes are based on cryptographic hash functions, the assumption that the hash function is pre-image resistant or collision resistant may not be sufficient, and we may be unable to find a sufficient assumption to prove the scheme secure.

To solve this problem we introduce idealised models. One very common such model is the random oracle model, which is described in Section 2.4.1. There are also other idealised models, such as the generic group model, where the adversary is given access to a randomly chosen encoding of a group and an oracle that executes the group operation.

Another way to create a model where we are able to prove security is to use trust assumptions. Here we assume that the parties have access to some trusted third party that perform some tasks honestly. One example of this is the common reference string model where we assume that all involved parties have access to some reference string that is sampled from some distribution. Additionally, when using a public key infrastructure (PKI) model, we assume that we have a trusted certificate authority that issues certificates honestly.

2.4.1 Random Oracle Model

In the random oracle model (ROM), we assume that the parties have access to (one or more) random oracles. This random oracle is a way to model a cryptographic hash function, H, as a truly random function. In the scheme, the parties can only evaluate H by querying (asking) the random oracle. The oracle can be thought of as a black box and works by returning a uniformly random value for each (unique) input.

We are not claiming that such a random oracle exists, but use it as a tool to prove the security of protocols that use cryptographic hash functions. First we prove the scheme to be secure in the random oracle model and then when we want to implement the scheme, we instantiate the random oracle with some appropriate cryptographic hash function H. In practice, security in ROM is a good heuristic for security in the standard model when the instantiation of the random oracle is a sufficiently strong hash function.

When we want to prove a cryptographic scheme to be post-quantum secure in ROM, we need to alter our model such that the adversary has quantum-access to the random oracle. This means that the adversary can query the random oracle with quantum states. The details of the quantum random oracle model (QROM) is presented in [3]. We need to make the distinction between ROM and QROM because when the random oracle is instantiated with a hash function, a quantum adversary will be able to evaluate it on quantum states. This may change the security properties of the scheme, and this is why we must include this ability in our model.

Note that ROM and QROM are strictly separated: There are protocols which are provably secure in ROM, but not in QROM (because it is not secure against adversaries that can evaluate the hash function on quantum states).

Chapter 3

Zero Knowledge Protocols

In many cases, Alice wants to be able to convince Bob that some statement is true. This can be to prove that she has access to some information, or that she computed some value in the way it was supposed to be computed. The statement Alice wants to prove has some information accompanying it that shows that the statement is true. This additional information is called a witness. There could be more than one witness for a statement, but in general we will talk about *the witness* meaning the one witness that exists or one or more of them in case there are several.

For example, if Alice wants to convince Bob that she has access to the discrete logarithm (to the base g, where g is some generator that is known to both Alice and Bob) of some value $x := g^a$, then a is the witness for this statement.

A trivial way to prove to Bob that we know the witness, is to simply reveal the witness. However, if we reveal the witness, it is obviously no longer secret, and in many cases, we don't want Bob to learn what the witness is. For example if the witness is some private key that we want to keep using, we do not want Bob to learn what it is. In fact, we do not want Bob to learn anything about our key, we only want Bob to learn that the statement is true. This property is what we call zero knowledge, and is introduced in more detail in Section 3.3. Before that, in Section 3.2 we will talk about what proofs of knowledge are and the properties we require from them.

3.1 Sigma protocols

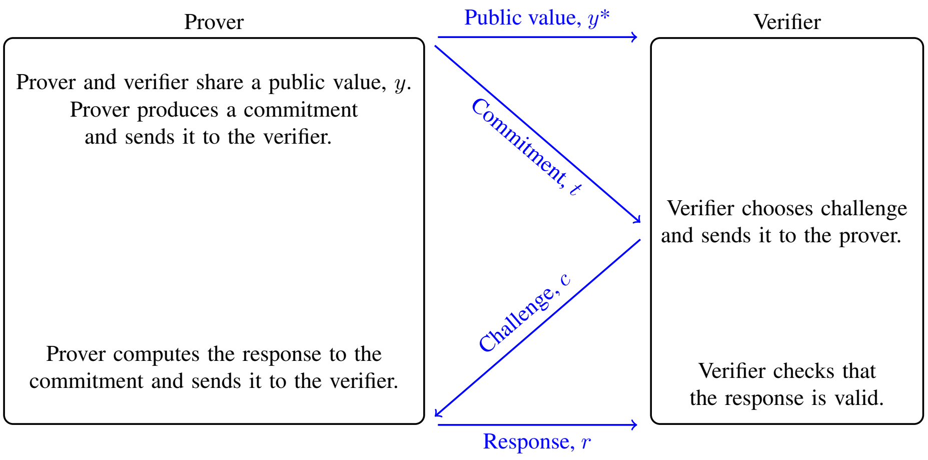
An identification scheme is the basis upon which we define the protocols in the rest of Chapter 3. The purpose of an identification scheme is for a prover to be able to prove its identity to the verifier. Typically, the prover will prove that it knows some secret information that only the claimed identity has access to.

We only consider identification schemes with a three-move structure (commitment, challenge, response), such that the scheme has algorithms KeyGen, Commit, and $\mathsf{Respond}$ run by the prover and Verify run by the verifier, and in addition, the verifier will choose a challenge for the prover to respond to.

Such an identification scheme with the three-move structure described, are called sigma protocols. The name comes from the visualisation of the message flow in the protocol as shown in figure 3.1, which looks like the greek letter sigma, Σ.

Definition 4. (Σ-protocols)
*A **sigma protocol** (Σ-protocol) is a three-move protocol with correctness. The three messages consist of:*

1. *the commitment, t,*
2. *the challenge, c,*
3. *and the response, r.*



Prover
Public value, $y*$
Verifier
Prover and verifier share a public value, y. Prover produces a commitment and sends it to the verifier.
Commitment, t
Verifier chooses challenge and sends it to the prover.
Challenge, c
Prover computes the response to the commitment and sends it to the verifier.
Verifier checks that the response is valid.
Response, r

Figure 3.1: Sigma protocol

The prover and verifier share a public value y. Figure 3.1 illustrates this by the first message: The Prover sends the public value to the Verifier. This does not usually happen in the protocol, but is included in the figure as it completes the Σ flow visually.

The protocol proceeds as follows: The Prover picks a random value, k, computes the commitment to this value, $t \leftarrow f(k)$ for some (one-way) function f, and sends the commitment to the Verifier. The Verifier chooses a challenge c and sends it to the Prover. The Prover computes the response, r, to the challenge c and sends it to the Verifier. The Verifier checks that the response r is valid given the commitment and the challenge as well as the public value.

Typically, this is repeated in order to increase confidence in the scheme. A cheating prover may be able to respond correctly to one or a few challenges by luck, but when the number of challenges grows and the prover is able to answer all of them consistently, it becomes increasingly unlikely that the prover is cheating.

The sigma protocols that we are interested in, are proofs of knowledge (Definition 5).

3.2 Proofs of knowledge

Informally, a proof of knowledge is a protocol executed by a prover and a verifier, allowing the prover to convince the verifier that they know some secret. For any cryptographic proof, we want the proof to be *complete*. This means that if a statement is true, then the verifier will always be convinced of this by an honest prover.

In order for a protocol to be a proof of knowledge, we require that there exists an extractor that finds the witness, given access to a (sufficiently) successful prover. This property tells us that any prover who is able to convince the verifier does have access to the secret and thus that any prover without access to the witness is unable to convince the verifier.

We make this a bit more formal: Let X be the set of statements and L be our language, that is the set of all true statements. Let W be the set of witnesses for the truth of the statements in our language. Then the relation $R \subseteq X \times W$ is a witness relation for L if $\forall x \in X(x \in L \Leftrightarrow \exists w \in W : (x, w) \in R)$. That is, a statement $x \in X$ is in our language L if and only if there is a witness $w \in W$ such that x and w are related: $(x, w) \in R$.

Let $P(x, w)$ be our prover with input a statement x and witness w such that $(x, w) \in R$. The prover will have a conversation with the verifier, V, and this conversation results in a proof π. This conversation will typically be a sigma protocol.

After receiving the last message of the sigma protocol, the verifier will check that the response is correct based on the commitment and challenge as well as the input x, so we can consider this response (the final message) to be the proof π. We will later talk about non-interactive protocols, where the proof is the whole content of the message from the prover to the verifier.

So we have that $V(x, \pi)$ is the verifier with input the statement x and the proof resulting from the sigma protocol, π. The verifier will output either Accept or Reject based on the result of the verification of the proof π.

Completeness here means that if $\pi \leftarrow P(x, w)$ (Where π is the final message in the sigma protocol executed by the prover and verifier) for some $(x, w) \in R$, then $V(x, \pi)$ outputs Accept. Completeness together with the existence of a witness extractor gives us that $\mathsf{Accept} \leftarrow V(x, \pi) \Leftrightarrow x \in L$, except with some small probability.

A typical example of a problem that we may want to prove statements about is the discrete logarithm problem, which is the basis of the $\mathsf{Diffie\text{-}Hellman\ algorithm}$. In the discrete logarithm problem, $L = \langle g \rangle$ since g generates the group, but for a random $x \in \langle g \rangle$ it is difficult to find a witness a such that $x = g^a$.

As we have already mentioned, a proof of knowledge means there exists a witness extractor. In order to prove that a protocol is a proof of knowledge, we construct such an extractor. Constructing this extractor shows that any (potentially malicious) prover who is able to convince the verifier, must have access to the witness.

An extractor is an efficient algorithm that finds the witness given access to a (sufficiently) successful prover. The success probability of the extractor depends on (and is large as a function of) the success probability of the prover in convincing the verifier:

Let $x \leftarrow \mathcal{D}_L$ where $\mathcal{D}_L$ is some distribution of our language L and let $Pr[P(x) \longleftrightarrow V(x) \Rightarrow \mathsf{Accept}] = \varepsilon$ (that is, the probability that the prover succeeds in convincing the verifier is ε). Then an extractor $\mathcal{E}$ with input x and P (i.e. with oracle access to the prover,

P) will output the witness w such that $(x, w) \in R$ with probability δ, where δ is large as a function of ε.

Our extractor $\mathcal{E}$ will not always be able to extract the witness from just *one* accepting conversation (one valid proof) provided by its oracle (the prover P). In addition to running the oracle several times on the same or on different input, we also give the extractor the ability to *rewind* its oracle: It can simulate the prover with some input x to get the first message m_1 and then feed the prover some challenge c_1, to which the prover responds with the second message m_2. At this point the extractor can return the oracle to the state it had after sending the first message m_1, before it received the challenge c_1. It can now feed the oracle some other challenge c_2 such that $c_1 \neq c_2$, and receive some m_2'. Since $c_1 \neq c_2$, we will have $m_2 \neq m_2'$. This way of achieving two (or more) different accepting proofs, but with some of the values equal (in this case the input, x, and any randomness chosen by the prover as well as m_1) is called *rewinding*.

We define a proof of knowledge formally in Definition 5.

Definition 5. (Proof of Knowledge)
Let $x \in L$ be a statement in a language and $w \in W$ a witness. A ***proof of knowledge*** *is a protocol $\Pi_{(P,V)}$ executed by a prover, P, and a verifier, V, such that*

- ***Completeness:*** *if $(x, w) \in R$, then $\Pi_{(P,V)}(x) \implies \mathsf{Accept}$, i.e. $(\pi \leftarrow P(x, w)) \implies (\mathsf{Accept} \leftarrow V(x, \pi))$, and*
- ***Existence of a witness extractor****: $\exists \mathcal{E}$ such that if $x \leftarrow \mathcal{D}_L$ and $Pr[\Pi_{(P,V)}(x) \implies \mathsf{Accept}] = \varepsilon$, then $Pr[w \leftarrow \mathcal{E}(x, P) : (x, w) \in R] = \delta$, where δ is large as a function of ε.*

The idea of a witness extractor plays a role similar to the *soundness* requirement for general cryptographic proofs. Soundness means that if a statement is false, then no cheating prover can convince the verifier that it is true (except with some small probability). The witness extractor gives us this soundness property for proofs of knowledge: if a prover is able to convince the verifier that they know the secret, then the extractor can find the secret, meaning that the prover does in fact have access to it (since it can just extract it).

We include the definition of soundness for completeness:

Definition 6. (Soundness)
Let $x \notin L$ be a false statement a prover P want to prove. Let $(\mathsf{Prove}, \mathsf{Verify})$ be a proof of knowledge protocol. Then $(\mathsf{Prove}, \mathsf{Verify})$ is ***sound*** *if for every prover P and for every false statement $x \notin L$ we have:*

$$Pr[\mathsf{Accept} \leftarrow \mathsf{Verify}(\mathsf{Prove}(x))] \leq 1 - \delta,$$

where δ is non-negligible.

Since $1 - \delta$ doesn't need to be negligible, but is significantly lower than 1, the natural way to make the probability of a false statement being proved small, if it is not already, is to repeat the process. For each repetition, the probability of a cheating prover succeeding gets smaller.

We can distinguish between proofs and arguments: a *proof* has statistical soundness, that is it is sound against a computationally unbounded prover, and an *argument* only has computational soundness, which is only sound against an efficient prover. However, we will not be distinguishing between these two in the coming chapters.

In addition to this general concept of soundness, there is a related concept called special soundness, or n-special soundness. Special soundness means that given two distinct accepting conversations on the same statement, we can extract a witness for the statement. n-special soundness is a generalisation of this idea, where you need n accepting conversations to find the witness.

Special soundness is a property that is common in sigma protocols. In general, special soundness of a protocol Π implies soundness of Π. For sigma protocols Π_Σ in particular, special soundness also implies that Π_Σ is a proof of knowledge which we will see in Theorem 1.

Definition 7. ((n-)special soundness)

Let $x \in L$ be a statement in our language and let w be a witness for x (so $(x, w) \in R$). Let Π_{Prove} be a sigma protocol. If Π_{Prove} has ***special soundness****, then given two accepting conversations (x, t, c, r) and (x, t, c', r') where $c' \neq c$ (and thus $r' \neq r$), there exists an extractor $\mathcal{E}$ that finds the witness w for x.*

As a generalisation of this, if Π_{Prove} has ***n-special soundness*** *then given n distinct accepting conversations for x there exists an extractor $\mathcal{E}$ that finds the witness w for x.*

Theorem 1.

Let $x \in X$ be a statement in our language L, $w \in W$ be a witness for the statement x, and R a relation such that $(x, w) \in R \Leftrightarrow w$ is a witness for x. Let $\Pi_{(P,V)}$ be a sigma protocol (that has completeness) with special soundness that takes as input some x. Then $\Pi_{(P,V)}$ is a proof of knowledge for w such that $(x, w) \in R$.

Proof. (Theorem 1) Completeness for the proof of knowledge follows from the completeness of the sigma protocol.

Let P be a prover that can reliably convince the verifier: $Pr[P(x) \longleftrightarrow V(x) \Rightarrow \mathsf{Accept}] = \varepsilon \geq 1/2$ for some $x \in L$. Then we have an extractor $\mathcal{E}$ that extracts the witness w such that $(x, w) \in R$, given oracle access to the prover P:

1. $\mathcal{E}$ simulates the prover on input x, and gets the first message. The prover P now has internal state Q.
2. $\mathcal{E}$ chooses a random challenge c_1, sends it to the prover (second message, m_2) and gets the third message m_3.
3. $\mathcal{E}$ rewinds the prover to state Q. $\mathcal{E}$ chooses another random challenge c_2 such that $c_1 \neq c_2$, sends it (m_2') to the prover and gets back the third message m_3' where we have $m_3' \neq m_3$ since $m_2 \neq m_2'$.
4. Since $\Pi_{(P,V)}$ has special soundness, $\mathcal{E}$ can now extract the witness.

Since the extractor needs two accepting conversations to extract the witness, its success probability is $\delta = \varepsilon^2$, the square of the success probability of the prover P. This is because the prover must succeed twice, for which it has success probability ε^2. Thus we have a witness extractor with success probability δ, which is sufficiently large.

The existence of an extractor $\mathcal{E}$ with sufficiently large success probability that extracts the witness, shows that $\Pi_{(P,V)}$ is a proof of knowledge. $\square$

In Example 7 we give a description of a proof of knowledge protocol and show that it is in fact a proof of knowledge.

Example 7. (Schnorr's protocol)

A prover P wants to convince the verifier V that it knows the discrete logarithm to base g of x, i.e. it has access to the witness a such that $x = g^a$. g is some generator of the group. If the prover does not want to simply reveal a to the verifier, it must give some proof of knowledge.

The language is $L = \{x \mid g^w = x\}$ for $w \in W$. Determining that $x \in L$ is trivial as for every x we have $x \in \langle g \rangle$. However, finding the witness w corresponds to solving the discrete logarithm problem. The relation in our proof of knowledge is $(x, w) \in R \iff x = g^w$.

Schnorr's protocol proceeds as follows:

1. *The prover chooses some randomness, r and commits to it: $t = g^r$. The prover sends the commitment t to the verifier.*
2. *The verifier chooses a random challenge c and sends it to the prover.*
3. *The prover computes the response to the challenge $s = r + ac$, and sends the response s to the verifier.*
4. *The verifier checks that $g^s = tx^c$, and accepts if this holds. Otherwise the verifier rejects the proof.*

The correctness of this protocol is easy to demonstrate: If $t = g^r$ and $x = g^a$ then $tx^c = g^r(g^a)^c = g^r g^{ac} = g^{r+ac} = g^s$, which is what was checked by the verifier. So if the statement is true, the verifier will accept.

To prove that this is a proof of knowledge we construct an extractor:

1. *Simulate the prover to output $t = g^r$.*
2. *Choose random challenge c_1 and input it to the prover. Get $s_1 = r + ac_1$ from the prover.*
3. *Rewind the prover. Choose a different challenge c_2 and input it to the prover. Get $s_2 = r + ac_2$ from the prover.*
4. *Output $(s_1 - s_2)(c_1 - c_2)^{-1} = ((r + ac_1) - (r + ac_2))(c_1 - c_2)^{-1} = (r + ac_1 - r - ac_2)(c_1 - c_2)^{-1} = a(c_1 - c_2)(c_1 - c_2)^{-1} = a$*

So the extractor was able to extract the witness a by running the prover two times on the same statement, by rewinding it.

3.3 Zero Knowledge

We now return to the zero knowledge property we sometimes want from our proofs: that the verifier doesn't learn anything about the witness, but only that the statement is true. We require three properties of zero knowledge proofs:

- Completeness: An honest verifier will be convinced by an honest prover if the statement is true.
- Soundness: No prover can convince a verifier that a false statement is true, except with some small probability.
- Zero knowledge: The protocol leaks no information apart from the fact that the statement is true. In particular, no information about the witness is obtained by the verifier.

In the case that our zero knowledge proof is a proof of knowledge, we require, like before, that there exists a witness extractor. In that case, the soundness requirement from above is just replaced with this new requirement as in Section 3.2.

The zero knowledge property is demonstrated by constructing a *simulator*. The simulator will be given access to the statement that should be proved, but not the witness, and will simulate an interaction between the verifier and the prover, that is the simulator will act like the prover in the protocol execution.

If the simulator is able to convince the verifier that the statement is true, the protocol is zero knowledge; since the simulator actually doesn't know the secret, the transcript of this interaction cannot reveal anything about the secret, and thus anything the verifier learns from the transcript it could have computed itself by just knowing the statement and the fact that it is true.

In particular, the verifier can only learn something from its *view*, which is all the values that are chosen by the verifier (its random coins) and the messages sent and received, that is, everything that the verifier sees during the protocol execution. So we say that if the view of the verifier interacting with the simulator is *indistinguishable* from the view of the verifier interacting with the prover, then the protocol is zero knowledge. We denote the view of the verifier and prover $view_V$ and $view_P$ respectively.

In the same way that we distinguish between proofs and arguments based on the abilities of the prover, we also distinguish between different notions of zero knowledge based on the verifiers abilities and behaviour. In any case, we cannot talk about zero knowledge with a computationally unbounded verifier, as it would be able to compute the witness from the common input.

Let $\Pi_{(P,V)}$ be a proof of knowledge for language L. Any execution $\Pi_{(P,V)}(x)$ of the protocol for some $x \in L$ results in a proof transcript as well as the output Accept or Reject. The transcript consists of the prover and verifiers transcripts interleaved. If any information about the witness should leak from $\Pi_{(P,V)}(x)$ the leakage must happen in the proof transcript, since there is no other information being shared between the parties.

If all the random variables in the proof transcript are uniformly random and independent from the input, then the verifier cannot learn anything from these values, and hence the proof transcript leaks no information. This is what we call perfect zero knowledge.

Another thing to consider is whether or not the verifier follows the protocol instructions. If the verifier tries to trick the prover to disclose information by not executing the protocol properly, we have a dishonest verifier. If a dishonest verifier is able to force the prover to produce a transcript which is unsimulatable, we no longer have perfect zero knowledge. We define a protocol to have honest verifier zero knowledge (HVZK) if the protocol has perfect zero knowledge when the verifier honestly executes the protocol instructions.

We also define a protocol $\Pi_{(P,V)}$ for L to be computational zero knowledge if for any $x \in L$, a proof transcript of $\Pi_{(P,V)}(x)$ can be simulated by an efficient algorithm with probability distributions that are computationally indistinguishable from the proof transcript.

Definition 8. (HVZK **proofs)**

Let $x \in L$ be a statement in our language and $w \in W$ a witness such that $(x, w) \in R$. An ***honest verifier zero knowledge*** *(*HVZK*)* ***proof*** *of x is a two-party protocol $\Pi_{(P,V)}$ such that the following properties are satisfied:*

1. *Completeness:* $(x, w) \in R \Rightarrow (\Pi_{(P,V)}(x) \implies \mathsf{Accept})$
2. *Soundness:* $(x, w) \notin R \Rightarrow (Pr[\Pi_{(P,V)}(x) \implies \mathsf{Accept}] \leq \varepsilon)$*, where ε is negligibly small.*
3. *There is a simulator, $\exists S$, such that for any $x \in L$, the output from $S(x)$ is indistinguishable from the verifier V's view from an interaction with the prover with input x, $\Pi_{(P,V)}(x)$, for any honest verifier V:* $view_V \leftarrow \Pi_{(P,V)}(x) \simeq view_V \leftarrow S(x)$

3.4 Non-interactive proofs

Many applications make the three-move sigma protocols impractical, and thus we want to achieve the same properties in a non-interactive protocol, that is a one-message protocol. One obvious advantage of non-interactive protocols is that the protocol does not require both parties to be online at the same time. The prover can simply generate the proof and send it, and the verifier can verify it at any time.

The verifier is convinced by the prover in sigma protocols because the prover is able to respond to a challenge of the verifiers choice. If the protocol is non-interactive, the verifier no longer controls the challenge, and thus we need some other mechanism to ensure that the prover cannot cheat. The most common method of achieving this is the Fiat-Shamir transform, introduced in section 3.4.1. In addition, Unruh's transform, introduced in section 3.4.2 also enables this in a different way with stronger security properties.

In addition to making our proofs non-interactive, we still want to achieve zero knowledge. However, it turns out that achieving zero knowledge (and in particular HVZK as we will want in later chapters) is not as straightforward for non-interactive proofs as it seemed for interactive proofs.

We attempt to define non-interactive zero knowledge (NIZK) like we did for HVZK by constructing a simulator that will indistinguishably simulate the prover in the protocol. However, there is a problem with this, as we are in one of two situations:

1. If the simulator can generate valid proofs for $x \in L$, then it is possible that it could also generate valid proofs for $x \notin L$. This breaks soundness.
2. If the simulator can generate valid proofs for $x \in L$, but not for $x \notin L$, then we can use the simulator to distinguish elements of our language L. This is computationally infeasible for many languages.

So we cannot have that the simulator can generate valid proofs for $x \in L$ since both of the above alternatives are not acceptable. In order to have a simulator that can do this, it needs som more information, that it will not have access to in the standard model. For this reason, we define NIZK in the random oracle model (ROM). Let $\Pi_{(P,V)}$ be the protocol we want to be NIZK. Let H be the hash function used in $\Pi_{(P,V)}$. We model H as a random oracle and give our simulator control over the access to this random oracle.

In order to show that $\Pi_{(P,V)}$ is NIZK, we need to show that our adversary (or potentially malicious verifier) cannot distinguish between the simulation and a real interaction with at prover. So we have two scenarios, and in each of them the adversary can ask two different things (proof queries and random oracle queries):

1. Scenario 0 - the real world:
   - A (justified) proof query $(x, w) \in R$: The challenger runs the prover part of the protocol $\Pi_{(P,V)}$, and returns $\pi \leftarrow P(x, w)$.
   - A random oracle query, u: The challenger chooses a random oracle, $\mathcal{O}$, and returns $c \leftarrow \mathcal{O}(u)$.
2. Scenario 1 - the simulated world:
   - A proof query $(x, w) \in R$: The challenger forwards the proof query without the witness and returns the simulated proof, $\pi' \leftarrow \mathcal{S}(y)$.
   - A random oracle query u: The challenger forwards the random oracle query and returns the simulator output, $c' \leftarrow \mathcal{S}(u)$.

Note that the proof queries made to the challenger from the adversary must include a witness, so the simulator will never attempt to prove false statements. So in any given interaction between the adversary and the challenger, the adversary could be in the real world scenario or in the simulated scenario, and its ability to distinguish between these two is the security notion we are interested in for NIZK proofs.

Definition 9. (NIZK **proofs of knowledge)**

Let $\mathcal{A}$ be an adversary against the NIZK *property, and let $\mathcal{D}ist_{\mathcal{A}}$ be the event that the adversary $\mathcal{A}$ correctly guesses which scenario it is in. A* ***non-interactive zero knowledge (*****NIZK*****) proof of knowledge****, in the random oracle model, is a one-message protocol $\Pi_{(P,V)}$ such that we have the following properties:*

1. *$\Pi_{(P,V)}$ is a proof of knowledge: It has completeness and there exists a witness extractor.*
2. *$\Pi_{(P,V)}$ is zero knowledge: The adversary cannot reliably distinguish between the real world and the simulation, so we have $|Pr[\mathcal{D}ist_{\mathcal{A}}] - \frac{1}{2}| \leq \varepsilon$ where ε is negligibly small.*

3.4.1 Fiat-Shamir transform

The Fiat-Shamir transform is a construction to convert a sigma protocol into a non-interactive signature sheme. As before, the prover will produce a commitment, t, but instead of the verifier choosing the challenge, the Prover computes the challenge itself by applying some pre-determined function H to the commitment and the message m. The challenge is then $c \leftarrow \mathsf{H}(t, m)$. Then the Prover computes the response r to the challenge c. The signature on the message m is then (c, r). The Verifier verifies the signature by recomputing the commitment t using the public value y, and the signature (r, s), checking that the challenge was computed correctly as $c = \mathsf{H}(t, m)$ and that the response r is valid for t and m.

The signature (c, r) depends on the message m since the challenge is a function of both the commitment and the message. If we model H as a random oracle, it will be just as difficult for an adversary without access to the secret key k_s to find a valid signature on the message m in this protocol as it would be in the original sigma protocol.

The security of the Fiat-Shamir transform relies on another result, called the Forking lemma (Lemma 1) [16].

Lemma 1. (Forking lemma)
Let Π_{H} be the signature system obtained by applying the Fiat-Shamir transform with hash function H to a sigma protocol Π that has special soundness. Let $\mathcal{A}$ be an adversary against the $\mathsf{EUF\text{-}CMA}$ security of the signature system Π_{H}. If $\mathcal{A}$ can, with non-negligible probability, find a valid signature (m, σ_1, h, σ_2), then, with non-negligible probability, $\mathcal{A}$ (with the same random tape and a different oracle) will output two valid signatures, (m, σ_1, h, σ_2) and (m, σ_1, h', σ_2') such that $h \neq h'$.

Now we can show that the Fiat-Shamir transform is secure in Theorem 2.

Theorem 2. (Fiat-Shamir transform security)
Let Π be a sigma protocol that has special soundness. Then the signature scheme made from applying the Fiat-Shamir transform to Π, modelling H as a random oracle, is secure against chosen message attacks ($\mathsf{EUF\text{-}CMA}$) in the random oracle model.

Proof. (Theorem 2) The security of the Fiat-Shamir transform depends on the Forking lemma. If a malicious prover $\mathcal{A}$ is able to forge a valid conversation (t, c, r), the Forking lemma shows us that $\mathcal{A}$ will be able to forge another valid conversation (t', c', r') where $t' = t$ and $c' \neq c$, with sufficient probability. From special soundness of the sigma protocol, $\mathcal{A}$ can then extract the witness from the two valid conversations. Thus we have an extractor that finds a witness given a (successful) malicious prover, and hence the Fiat-Shamir transform yields a proof of knowledge. Since the proof of knowledge depends on the message, m, this proof of knowledge gives us a $\mathsf{EUF\text{-}CMA}$ secure signature scheme. $\square$

In the quantum setting, the security of the Fiat-Shamir transform which is based on the Forking lemma collapses. This is because the rewinding that the Forking lemma relies on cannot be done in the quantum setting, so the Forking lemma does not hold. Since this way of proving the security of the Fiat-Shamir transform collapses, we consider other methods of achieving non-interactivity.

3.4.2 Unruh's transform

Unruh's transform [18] is a non-interactive transform that is an alternative to Fiat-Shamir transform. We are interested in Unruh's transform because, unlike the Fiat-Shamir transform, it is secure in QROM and it has online extractability. Unruh's transform takes as input a sigma protocol with n-special soundness and outputs a non-interactive proof system, and in fact a EUF-CMA secure signature scheme, that is secure against quantum adversaries in QROM. If the sigma protocol is a zero knowledge protocol (specifically, it has HVZK), then the resulting protocol is NIZK (see Defintion 9). So Unruh's transform preserves the HVZK property of the sigma protocol as well as the security of the proof of knowledge in QROM. We define online extractability (and the stronger notion simulation-sound online extractability) in Definition 10 (Definition 11 respectively).

Unruh's transform works by the Prover doing the following:

1. Prover takes input (x, w), and will create $m \times n$ proofs $(com_i, ch_{i,j}, resp_{i,j})$.
2. For each of the proofs, the Prover commits to the responses.
3. The Prover computes the challenge by hashing a number of values.
4. Last, the Prover reveals the response in some of these proofs, based on the challenge computed in the previous step. There will be exactly one response for each distinct commitment, such that the Prover never reveals two accepting conversation for the same commitment.
5. The Prover sends π to the Verifier, which is a list containing the commitments, challenge and commitment to the responses for all the $m \times n$ proofs it created, as well as the responses for the proofs determined by the challenge.

The Verifier takes as input (x, π). It computes the challenge like the Prover did, checks that the challenges in all the proofs are pairwise distinct, that all the (opened) conversations are accepting and that the commitments to the (opened) responses are correctly computed. If all of the checks succeed, then the Verifier accepts.

There are five subroutines involved in Unruh's transform. The Prover algorithm uses two, the Verifier algorithm uses one. In addition they both use two one-way functions:

1. $\mathsf{Prove}^1_\Sigma(x, w)$ is used to generate challenges to the commitments in the $m \times n$ proofs.
2. $\mathsf{Prove}^2_\Sigma(ch_{i,j})$ is used to compute responses in the $m \times n$ proofs.
3. $\mathsf{Verify}_\Sigma(x_i, com_i, ch_{i,j}, resp_i)$ is used to verify the m proofs that have responses.
4. $\mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$ is used to generate the challenge, that is which responses should be included in the final proof.
5. $\mathsf{G}(resp_{i,j})$ is used to generate commitments to the $m \times n$ responses.

In Figures 3.2 and 3.3, the Prover and Verifier (respectively) algorithms for Unruh's transform are showed. Example 8 shows what the protocol does in a toy example with $m = 4, n = 3$.

```
Prove (x, w):
for $i = [1, m]$ do
    $com_i \leftarrow P^1_\Sigma(x, w)$
    for $j = [1, n]$ do
        $ch_{i,j} \xleftarrow{\$} N_{ch} \setminus \{ch_{i,1}, \dots, ch_{i,j-1}\}$
        $resp_{i,j} \leftarrow P^2_\Sigma(ch_{i,j})$
    end
end
for $i = [1, m]$ do
    for $j = [1, n]$ do
        $h_{i,j} := \mathsf{G}(resp_{i,j})$
    end
end
$J_1 \parallel J_2 \parallel \cdots \parallel J_m := \mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
return $\Pi := ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$
```

Figure 3.2: Unruh's transform prover algorithm

```
Verify $(x, \Pi)$:
$J_1 \parallel J_2 \parallel \cdots \parallel J_m := \mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
for $i = [1, m]$ do
    Check that $ch_{i,1}, \dots, ch_{i,n}$ are pairwise distinct.
end
for $i = [1, m]$ do
    Check that $V_\Sigma(x_i, com_i, ch_{i,J_i}, resp_i) = 1$
end
for $i = [1, m]$ do
    Check that $h_{i,J_i} = \mathsf{G}(resp_i)$
end
if All checks succeed then
    return Accept
else
    return Reject
end
```

Figure 3.3: Unruh's transform verifier algorithm

Before we prove the security of Unruh's transform in Theorem 3, we define formally what we mean by online extractability (Definition 10), as well as simulation-sound online extractability (Definition 11) which is the security notion Unruh's transform satisfies. We will not include the proof for the simulation-sound online extractability, but include the definition so that it is clear what is required. The full proof for the simulation-sound online extractability can be found in the original paper presenting Unruh's transform [18].

In the classical sense, online extractability means that the witness can be extracted by looking only at the proofs generated by the (possibly malicious) prover and the oracle queries it made. In the quantum setting, it is not possible to define the list of oracle queries because that would mean measuring the oracle input, which disturbs it. Instead of this, we give the extractor the description of the oracle H that was output by the simulator $\mathcal{S}_{init}$, showed in Figure 3.5.

Definition 10. (Online extractability)

Let $x \notin L$ be a statement that is not in our language. Let H be the oracle output by the simulator $\mathcal{S}_{init}$. A non-interactive proof system $\Pi_{(P,V)}$ is online extractable with respect to $\mathcal{S}_{init}$ if and only if there is an extractor $\mathcal{E}$ such that for any efficient quantum adversary $\mathcal{A}$ we have:

$$Pr[\mathsf{Accept} \leftarrow V(x,\pi) \textit{ and } (x,w) \notin R, \textit{ given } (x,\pi) \leftarrow \mathcal{A}(), w \leftarrow \mathcal{E}(\mathsf{H},x,\pi)] \leq \varepsilon,$$

where ε is negligibly small.

Online extractability implies that an adversary cannot produce a proof (that will be accepted by a verifier) for a statement that it knows no witness for, because an extractor can extract a witness from the proof. However, an adversary may still be able to take a valid proof π_1 for statement x_1 and turn it in to a valid proof π_2 for a different statement x_2 (even if it does not know a witness for x_2, as long as a witness for x_2 is efficiently computable from a witness for x_1). This is what we call malleability. The stronger definition simulation-sound online extractability avoids this problem. This is the requirement that extraction of a witness from any proof generated by an adversary $\mathcal{A}$ should be successful even if $\mathcal{A}$ has access to simulated proofs.

Definition 11. (Simulation-sound online extractability)

Let $x \notin L$ be a statement that is not in our language. Let $\{\pi\}_{\mathcal{S}_P}$ be the set of all proofs output by the simulator $\mathcal{S}_P$, $(x,\pi) \leftarrow \mathcal{A}()$ is the proof output by the adversary $\mathcal{A}$ and $w \leftarrow \mathcal{E}(\mathsf{H},x,\pi)$ is the witness extracted by the extractor $\mathcal{E}$ from the proof output by $\mathcal{A}$.
A non-interactive proof system, $\Pi_{(P,V)}$ is simulation-sound online extractable with respect to simulators $\mathcal{S}_P$ and $\mathcal{S}_{init}$ if and only if there is an efficient extractor $\mathcal{E}$ such that for any efficient quantum adversary $\mathcal{A}$, we have:

$$Pr[\mathsf{Accept} \leftarrow V(x,\pi), (x,\pi) \notin \{\pi\}_{\mathcal{S}_P} \textit{ and } (x,w) \notin R] \leq \varepsilon,$$

where ε is negligibly small.

Theorem 3. (Unruh's transform)

Let Π_U be the proof system obtained by applying the Unruh's transform to a sigma protocol Π that has completeness, special soundness and HVZK*. Then Π_U is a secure*

NIZK *proof system with simulation-sound online-extractability with respect to the simulators $\mathcal{S}_{\Pi_U}$, $\mathcal{S}_{init}$ and extractor $\mathcal{E}_{\Pi_U}$ (Figures 3.4, 3.5, and 3.6 respectively).*

Proof sketch. (Theorem 3) We will give a sketch to the proof of the compeleteness and HVZK of Π_U.

Online extractability: The extractor $\mathcal{E}_{\Pi_U}$ (Figure 3.6) is constructed in order to prove this property. We omit the proof for online extractability, which is included in detail in [18].

Completeness: Since Π has completeness, $\mathsf{Verify}(x, com_i, ch_{i,j}, resp_{i,j}) = 1$ for all i, j. If all the proofs were computed according to the protocol, all the verifiers checks will succeed. So Π_U is correct.

Zero knowledge: Simulators $\mathcal{S}_{\Pi_U}$ (Figure 3.4) and $\mathcal{S}_{init}$ (Figure 3.5) is constructed to show HVZK, and then the proof continues using a sequence of games.

Game 1: The real world. The protocol runs as it should.

Game 2: We modify the prover: it chooses a random challenge $J_1, \dots, J_m$ instead of getting it from the random oracle H, and programs the random oracle to return those values. Game 2 is indistinguishable from game 1 for an attacker.

Game 3: For each i, now the prover produces the conversation belonging to the challenge, $(com_i, ch_{i,j}, resp_{i,j})$ first, and then all the other ones. Changing the order in which the conversations are produced changes nothing. Since the challenges $ch_{i,j}$ are chosen uniformly at random (but being pairwise distinct), game 3 is indistinguishable from game 2.

Game 4: Now we let the commitments $h_{i,j}$ that will never be opened be picked at random instead of using G. An attacker who can distinguish between games 3 and 4 can distinguish between G and a random function, so this can happen only with negligible probability.

Game 5: Now the prover doesn't compute the responses for the conversations that will not be opened. This doesn't change anything from the verifiers perspective, so game 5 is indistinguishable from game 4.

Game 6: In this game, the honestly generated proof is replaced by one that is produced by the simulator $\mathcal{S}_{\Pi_U}$. Since Π is HVZK, game 6 is indistinguishable from game 5. Note that the prover no longer uses the witness w to create the proofs.

Game 7: We replace the prover by the simulator $\mathcal{S}_{\Pi_U}$. This doesn't change anything as the simulator does exactly what the prover in game 6 does. So game 7 is indistinguishable from game 6.

Game 8: We replace the oracles H and G randomly chosen among oracles, to oracles that are constructed in a specific way by the simulator $\mathcal{S}_{init}$. Both the old pair and the new pair of oracles are indistinguishable from random, so this doesn't change anything. So game 8 is indistinguishable from game 7.

Since game 8 is indistinguishable from game 1 (except with some negligible from some of the game hops), the protocol Π_U is HVZK (in the random oracle model).

□

```
for $i = [1, m]$ do
    Choose random $J_i$ from $\{1, \dots, n\}$
    Get the accepting conversation $(com_i, ch_{i,J_i}, resp_{i,J_i})$ from the simulator,
      $\mathcal{S}_{\Pi}$, for the zero knowledge sigma protocol (proof of knowledge), $\Pi$.
    For each $j \in \{1, \dots, n\}$ except $j = J_i$, choose a random challenge $ch_{i,j}$ that
      is not equal to any previously used challenge.
for $i = [1, m]$ do
    Commit to the responses from $\mathcal{S}_{\Pi}$: $h_{i,J_i} \leftarrow \mathsf{G}(resp_{i,J_i})$
    For each $j \in \{1, \dots, n\}$ except $j = J_i$, choose a random value to act as a
      "commitment", $h_{i,j}$, for the responses that will not be opened.
Program the random oracle H to return the correct challenge:
  $\mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}) := J_1 \parallel \cdots \parallel J_m$
return $\pi := ((com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}, (resp_{i,J_i})_i)$
```

Figure 3.4: Simulator $\mathcal{S}_{\Pi_U}$.

```
Parameters: Upper bounds $q_{\mathsf{G}}, q_{\mathsf{H}}$ on the number of queries to G and H,
  respectively. Upper bound on length of input to H. Embedding $\iota_l$.
$p_{\mathsf{G}} \xleftarrow{\$} GF(2^{l_{\text{resp}}})[X]$ with $\partial_{p_{\mathsf{G}}} \leq 2q_{\mathsf{G}} - 1$
$p_{\mathsf{H}} \xleftarrow{\$} GF(2^{l})[X]$ with $\partial_{p_{\mathsf{H}}} \leq 2q_{\mathsf{H}} - 1$
Construct circuits G, H:
$\mathsf{G}(x) := p_{\mathsf{G}}(x)$ for $x \in \{0,1\}^{l_{\text{resp}}}$
$\mathsf{H}(x) := p_{\mathsf{H}}(\iota_l(x))_{1 \dots t \log m}$ for $x \in \{0,1\}^{l}$
return descriptions of H and G
```

Figure 3.5: Simulator $\mathcal{S}_{init}$ for Unruh's transform online extractability. See [18] for details.

```
$\mathcal{E}_{\Pi_U}(\mathsf{G}, \mathsf{H}, x, \pi = ((com_i), (ch_{i,j}), (h_{i,j}), (resp_{i,j})))$:
The extractor gets $J_i \parallel \cdots \parallel J_m \leftarrow \mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j})$
for $i = [1, m]$ do
    for $j = [1, n]$ do
        for each $resp' \in \mathsf{G}^{-1}(h_{i,j})$ do
            if $\mathsf{Verify}(x, com_i, ch_{i,j}, resp) = 1$ then
                return $\mathcal{E}_{\Pi}(x, com_i, ch_{i,J_i}, resp_i, ch_{i,j}, resp')$
```

Figure 3.6: Extractor $\mathcal{E}_{\Pi_U}$.

Note that the original proof for the security of Unruh's transform assumes that the sigma protocol it is applied to has 2-special soundness. In PICNIC, a sigma protocol that has 3-special soundness (and *not* 2-special soundness) is used. In order to be able to use

Unruh's transform, it must be proven secure for sigma protocols with 3-special soundness (or more generally, with n-special soundness). This has been done in [9] and we omit the details of this proof as it closely resembles the original proof.

Unruh's transform gives a substantial overhead to the protocols it is applied to, since the transformation involves making $m \times n$ proofs. Given a security parameter it is possible to optimise the values of m and n to achieve the security required while minimising the computational cost. In PICNIC, Unruh's transform is applied to ZKB++ (see Section 6.3).

In ZKB++, the challenge space is limited to $\{1, 2, 3\}$, so we cannot have $n > 3$. Therefore it is natural to set $n = 3$. In this case, we can find the minimum m to achieve the required security level. In the paper introducing PICNIC [5], this is described in detail. For 128-bit post-quantum security, we set $m = 438$. This is double what would be required in the classical setting, due to algorithms designed for quantum computers that reduce the time to break the system for an adversary.

Example 8. (Unruh's transform)

$m = 4, n = 3$

Prover:

Sample 4 commitments: com_1, com_2, com_3 *and* com_4 *For each commitment, create 3 accepting conversations:*

- *For* com_1*:*
  - *Sample three challenges:* $ch_{1,1}, ch_{1,2}, ch_{1,3}$.
  - *Compute responses to each of the challenges:* $resp_{1,1} resp_{1,2}, resp_{1,3}$.
- *For* com_2*:*
  - *Sample three challenges:* $ch_{2,1}, ch_{2,2}, ch_{2,3}$.
  - *Compute responses to each of the challenges:* $resp_{2,1} resp_{2,2}, resp_{2,3}$.
- *For* com_3*:*
  - *Sample three challenges:* $ch_{3,1}, ch_{3,2}, ch_{3,3}$.
  - *Compute responses to each of the challenges:* $resp_{3,1} resp_{3,2}, resp_{3,3}$.
- *For* com_4*:*
  - *Sample three challenges:* $ch_{4,1}, ch_{4,2}, ch_{4,3}$.
  - *Compute responses to each of the challenges:* $resp_{4,1} resp_{4,2}, resp_{4,3}$.

For each response, for each commitment, commit to the response:

$$h_{1,1}, h_{1,2}, h_{1,3}, h_{2,1}, \ldots, h_{4,2}, h_{4,3}.$$

Compute the challenge by hashing:

$$J_1 \parallel J_2 \parallel J_3 \parallel J_4 := \mathsf{H}(x, (com_i)_i, (ch_{i,j})_{i,j}, (h_{i,j})_{i,j}),$$

where J_i *are integers* $1 \leq J_i \leq 3$.

The challenge points out one response for each commitment to reveal, that is exactly one from $(resp_{1,1}, resp_{1,2}, resp_{1,3})$ *which are the responses belonging to the first commitment* com_1*, and similarly for* com_2, com_3 *and* com_4*.*

In this example, let $J_1 = 1, J_2 = 3, J_3 = 2, J_4 = 2$.

Send to the Verifier: $\pi :=$ *contents of table*			
com_1	com_2	com_3	com_4
$ch_{1,1}, ch_{1,2}, ch_{1,3}$	$ch_{2,1}, ch_{2,2}, ch_{2,3}$	$ch_{3,1}, ch_{3,2}, ch_{3,3}$	$ch_{4,1}, ch_{4,2}, ch_{4,3}$
$resp_{1,1}$	$resp_{2,3}$	$resp_{3,2}$	$resp_{4,2}$
$h_{1,1}, h_{1,2}, h_{1,3}$	$h_{2,1}, h_{2,2}, h_{2,3}$	$h_{3,1}, h_{3,2}, h_{3,3}$	$h_{4,1}, h_{4,2}, h_{4,3}$

Verifier:

- *Compute the challenge:* $J_1 = 1, J_2 = 3, J_3 = 2, J_4 = 2$.
- *Check that all challenges* $(ch_{i,j})_{i,j}$ *are pairwise distinct.*
- *Check that the conversations cosrresponding to these challenges are accepting:*
  - $(com_1, ch_{1,1}, resp_{1,1})$,
  - $(com_2, ch_{2,3}, resp_{2,3})$,
  - $(com_3, ch_{3,2}, resp_{3,2})$, *and*
  - $(com_4, ch_{4,2}, resp_{4,2})$.
- *Check that the commitments to the responses corresponding to these challenges are correct:*
  - $h_{1,1} = \mathsf{Commit}(resp_{1,1})$,
  - $h_{2,3} = \mathsf{Commit}(resp_{2,3})$,
  - $h_{3,2} = \mathsf{Commit}(resp_{3,2})$, *and*
  - $h_{4,2} = \mathsf{Commit}(resp_{4,2})$.
- *If all the above checks are successful, output* Accept. *Otherwise output* Reject.

Chapter 4

One-Way Functions

In this chapter we will introduce one-way functions and block ciphers, and in particular see how we can construct a one-way function from a block cipher. We will also see two examples of block ciphers: LowMC which is used as a one-way function in the PICNIC signature scheme and AES which is the most commonly used block cipher today, and also is used in some of the other signature schemes with the same construction as PICNIC.

4.1 One-way functions

We start by defining one-way functions. These are functions that are easy to compute on any input, but hard to invert given the image of some input.

Definition 12.
A function $f(\cdot)$ is one-way if:

1. *it can be computed by an efficient algorithm, and*
2. *any efficient algorithm F that attempts to compute an inverse succeeds only with negligible probability, i.e. for all F,*

$$Pr[f(F(f(x))) = f(x)] < \varepsilon,$$

where ε is negligibly small and the probability is over the uniform distribution of all possible function inputs x.

Note that the existence of a true one-way function has not been proved, but there are functions that we consider to be one-way because we currently do not know of any efficient algorithm that is able to invert them. Examples of such functions are:

- Multiplication of two primes: $f(p, q) = p \times q$ for which p and q are primes. The inverse function, factorisation of numbers, is considered hard (in the classical setting, this is not one-way in the quantum setting as we can efficiently factor numbers with quantum computers).

- Modular exponentiation: $f(g, a) = g^a = b$. Inverting this requires finding the discrete logarithm $a = log_g(b)$, which is considered hard (again, in the classical setting).
- Cryptographically secure hash functions, such as SHA-256.

4.2 Pseudo-random permutations

In order to introduce block ciphers, we need to be familiar with pseudorandom permutations (PRPs).

The $G^{\text{PRP-sec}}_{(\pi,\pi^{-1})}(\mathcal{A})$ experiment with adversary $\mathcal{A}$ proceeds as follows:

1. Sample $b \xleftarrow{\$} \{0,1\}$ and $k \xleftarrow{\$} \mathcal{K}$. Let $C_0 = C_1 = C_2 = \emptyset$.
2. When the adversary $\mathcal{A}$ sends a query $s \in \{0,1\}^n$, then:
   (a) If $s \notin C_1$, sample $s' \xleftarrow{\$} S \setminus C_2$, then add (s, s') to C_0, s to C_1 and s' to C_2.
   (b) If $s \notin C_2$, sample $s'' \xleftarrow{\$} S \setminus C_1$, then add (s'', s) to C_0, s'' to C_1 and s to C_2.
   (c) The experiment finds $u'_1, u''_1 \in S$ such that $(s, u'_1) \in C_0$ and $(u''_1, s) \in C_0$.
   (d) The experiment computes $u'_0 \leftarrow \pi^{-1}(k, s)$ and $u''_0 \leftarrow \pi(k, s)$.
   (e) The experiment sends (u'_b, u''_b) to $\mathcal{A}$.
3. Eventually, the adversary $\mathcal{A}$ outputs $b' \in \{0,1\}$.
4. The experiment outputs 1 if and only if $b' = b$.

Define the advantage of the adversary $\mathcal{A}$ in the PRP security experiment $G^{\text{PRP-sec}}_{(\pi,\pi^{-1})}(\mathcal{A})$ to be

$$Adv^{\text{PRP-sec}}_{(\pi,\pi^{-1})}(\mathcal{A}) = |Pr[G^{\text{PRP-sec}}_{(\pi,\pi^{-1})}(\mathcal{A}) = 1] - \frac{1}{2}|$$

Figure 4.1: PRP security experiment: $G^{\text{PRP-sec}}_{(\pi,\pi^{-1})}(\mathcal{A})$

A permutation is a bijection on our set of values, in our case we will be interested in all bit strings of a particular length: $\mathcal{S} = \{0,1\}^n$. A pseudorandom permutation family is a collection of permutations (π, π^{-1}) where you can choose a specific permutation by choosing a key k from the set of all possible keys of a particular length, $\mathcal{K} = \{0,1\}^m$, such that $\pi^{-1}(k, \pi(k,x)) = x$ for all $x \in \mathcal{S}$. We want each of these permutations to be indistinguishable from a random permutation.

Definition 13. (Pseudorandom permutations)
A pseudorandom permutation (family), is a mapping

$$\begin{aligned} \pi : \mathcal{K} \times \mathcal{S} &\to \mathcal{S}, \\ (k,m) &\mapsto \pi(k,m), \text{ such that:} \end{aligned}$$

1. *for each $k \in \mathcal{K}$, $\pi(k,\cdot)$ is a bijection on $\mathcal{S}$, and*
2. *for each $k \in \mathcal{K}$, $\pi(k,\cdot)$ and $\pi^{-1}(k,\cdot)$ are efficiently computable.*

The security of a PRP (π, π^{-1}) captures if it is indistinguishable from a random permutation. The adversary $\mathcal{A}$ is playing the experiment by making queries for values $s = \{0,1\}^n$. The experiment replies with either the actual PRP π and its inverse π^{-1} evaulated in s or with a random permutation evaluated in s. Eventually the adversary has to guess if it was interacting with the PRP or with a random permutation, and if if guesses correctly, it wins the game.

Figure 4.1 shows the security experiment for PRPs.

4.3 Block ciphers

A block cipher is a pseudorandom function family (π, π^{-1}). Block ciphers are traditionally used to build symmetric-key encryption schemes, meaning that the same key is used to encrypt and decrypt data. A key k is chosen uniformly at random from the key space, $\mathcal{K}$. The domain and the range of the block cipher are the same, and we call it the message space, $\mathcal{S}$.

The block cipher (family) encrypts and decrypts messages:

$$\begin{aligned} \pi, \pi^{-1} &: \mathcal{K} \times \mathcal{S} \to \mathcal{S} \\ \pi &: (k,m) \mapsto \pi(k,m) \\ \pi^{-1} &: (k,m) \mapsto \pi^{-1}(k,m), \end{aligned}$$

such that $\pi^{-1}(k, \pi(k,m)) = m$.

For a block cipher to be secure, the advantage of an adversary $\mathcal{A}$ against the PRP security of (π, π^{-1}) must be negligible.

Block ciphers are usually what we call iterated block ciphers. The block cipher takes a fixed-sized key, fixed sized input blocks and outputs blocks of the same size as the input blocks. This is done by repeatedly evaluating an invertible transformation called the *round function*. This round function describes the permutation family.

In the PICNIC construction, block ciphers are used only as one-way functions, so we want to construct a one-way function from a block cipher family, (π, π^{-1}). Clearly π :

$\mathcal{K} \times \mathcal{S} \to \mathcal{S}$ is not a one-way function. In fact, we know the inverse function π^{-1}, and given m and k, we can find m' such that $\pi(k, m') = m$ by computing $\pi^{-1}(k, m) = m'$.

However, for a fixed input $m \in \mathcal{S}$, consider the function

$$\begin{aligned} f_m : \mathcal{K} &\to \mathcal{S} \\ k &\mapsto c := \pi(k, m). \end{aligned}$$

For block ciphers that we want to use, it is reasonable to assume that it is hard to find a pre-image for f_m. This means that it is hard to find the block cipher key given an input-output pair, (m, c). If an efficient adversary $\mathcal{A}$ can find k, then $\mathcal{A}$ can distinguish the PRP $(\pi(k, \cdot), \pi^{-1}(k, \cdot))$ from a random permutation. This means that the block cipher was not secure. Hence, if we (believe we) have a secure block cipher, f_m is a one-way function for all messages m.

For the PICNIC signature scheme, we are interested in block ciphers that can be described using the simplest algebraic circuit possible. Due to this, we move on to introduce algebraic circuits and later to describe a specific block cipher that has particularly nice properties in this regard.

4.4 Algebraic circuits

In order to compute a function or execute an algorithm with some input, we can model the function or algorithm as an algebraic circuit. A circuit is a directed graph that has a tree-like structure. The root node represents the output and the leaf nodes represent the input. All the internal nodes represent some algebraic operation. We call these nodes *gates*. The graph representing the circuit may not be a tree as one input node may lead to more than one internal node (the input value is used in more than one algebraic operation in the evaluation of the circuit).

Figure 4.2 shows a simple example of what such a circuit with four inputs and one output may look like. We can see in the figure that all the nodes are labelled by either some input, output or an algebraic operation. The circuit evaluation is from the top to the bottom and the result is the output at the root node.

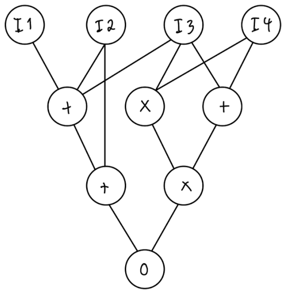


Figure 4.2: Simple algebraic circuit

Algebraic circuits have varying complexity depending on its depth and the number and types of operations on its internal nodes. This impacts what circuits are suitable for a specific application.

4.5 LowMC

LowMC [1] is the block cipher used as a one-way function in PICNIC. LowMC is a non-standard block cipher (as opposed to AES, which is introduced in Section 4.6). It is used because it has particularly low multiplicative complexity, which is also the reason it is named the way it is.

LowMC is a SPN type block cipher. The substitution-permutation network (SPN) is an important type of iterative block cipher, where the round function consists of a substitution stage followed by a permutation stage. The substitution stage is the evaluation of a substitution box (S-box). Similarly the permutation stage evaluates a permutation box (P-box).

For each round, you obtain a round key from the key (using some operations, e.g. S-boxes and P-boxes) and combine the result with the output from the round function using some group operation (e.g. addition or multiplication in the field).

LowMC is a family of block ciphers designed to minimise the multiplicative complexity of its circuit. The purpose of the cipher is to use it within multiparty computation (MPC), fully homomorphic encryption (FHE) and zero knowledge proofs (ZK). For these applications, non-linear operations (such as multiplication) is much more expensive in terms of computational and communication costs compared to linear operations (such as addition).

This is because while linear operations occur locally, non-linear operations require communication between the parties in MPC (and it increases noise significantly in FHE, but this will not be relevant for our application as we do not use FHE).

The question when designing the cipher was ”what is the minimum number of multiplications for building a secure block cipher [1]? In particular, the multiplications will be in $\mathbb{F}_2$. Note that in $\mathbb{F}_2$, multiplication is the same as the binary bit operation AND, and addition is the same as XOR. So what we want is to minimise the number of AND gates.

In the instantiation of LowMC, there are several parameters that can be independently chosen:

- Block size, n
- Key size, k
- Number of S-boxes in substitution layer, m
- Security parameter, d

In order to reduce the multiplicative complexity (MC), the number of S-boxes that are applied in parallel can be reduced. The reduction in security caused by this is mitigated by higher complexity in the linear layer. The LowMC round function is composed of four subroutines:

$$\mathsf{LowMC}_{\mathsf{Round}}(i) = \mathsf{KeyAddition}(i) \circ \mathsf{ConstantAddition}(i) \circ \mathsf{LinearLayer}(i) \circ \mathsf{S\text{-}BoxLayer}$$

Cipher	Key size	Block size	ANDdepth	ANDs/bit
LowMC	128	256	12	8.85
AES-128	128	128	40	43

Table 4.1: Comparison of cipher parameters

Each S-box is applied to 3 bits, so with m S-boxes, the first $3m$ bits of the current state is affected. If this is less than the whole block size, the remaining bits are unchanged. Figure 4.3 shows a visual represenation of the LowMC round function.

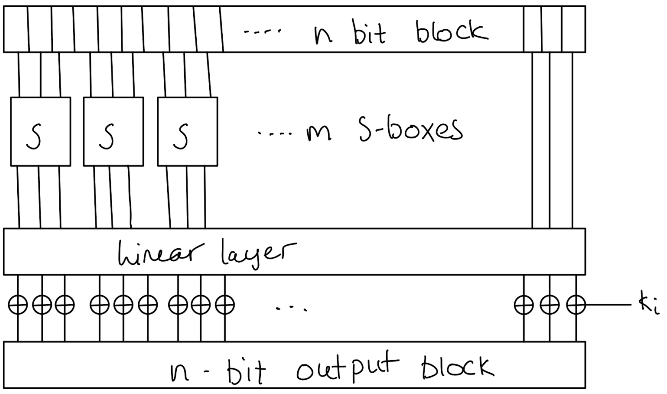


Figure 4.3: LowMC round function

Decryption is done by inverting each of the steps for encryption and doing them in the opposite order - like moving backwards in the encryption circuit.

In table 4.1 the parameters of LowMC are compared to AES-128. It is clear that if the number of AND gates and their depth dominate the computational cost, then LowMC will be cheaper to use.

LowMC is the block cipher used to instantiate the one-way function required in PICNIC (see Chapter 7), and this is due to its particular suitability for MPC. In Chapter 6 we will introduce how this is relevant in the MPC context. In PICNIC, LowMC is used only as a one-way function, so only the PRP security of LowMC is relevant to the security of PICNIC.

4.6 AES

One of the most common (family of) block ciphers in use today is the Advanced Encryption Standard (AES). It is the clear alternative to LowMC for use in PICNIC, but due to the increased multiplicative complexity of the algebraic circuit for AES compared to LowMC, the latter was chosen. However, other instantiations of the general approach to designing signature scheme that we introduced in Section 1.1 have opted for AES to instantiate the one-way function needee. BBQ [8] and BANQUET [2] are examples of such signature schemes.

Like LowMC, AES is also a SPN type cipher. AES has 128-bit block length and there is the option between three key lengths: 128 bits, 192 bits or 256 bits. The length of the keys affect the number of rounds the cipher has, as well as the key schedule. Each round, the *state*, which is a 4-by-4 array of bytes is modified by the round function. The initial state is the plaintext block, which is 128 bits (16 bytes).

The AES round function has 4 stages:

1. AddRoundKey : A 128 bit round key is derived and the state is updated by XOR-ing it with the round key.

2. SubBytes: Each byte of the state is replaced by another byte according to a fixed lookup table S. This is the S-box of the cipher and it is a bijection over $\{0,1\}^8$.

3. ShiftRows: The bytes in each row are shifted to the left such as: row 1 remains the same, row 2 is shifted one place, row 3 is shifted 2 places, row 4 is shifted 3 places.

4. MixColumns: An invertible transformation is applied to the bytes in each column.

Because the last three stages of the round function do not depend on the key, the final round replaces stage 4 with stage 1.

The AES cipher is the result of the NIST standardisation process that began in 1997, to standardise a new block cipher to replace previous ones. Having been part of this process, the cipher has been subject to intense scrutiny since then without any practical attacks against it. Due to this, the cryptographic community is fairly confident that it is secure and a good choice for any scheme that needs a strong pseudorandom permutation, and thus also confident that the resulting one-way function is secure.

Chapter 5

Multi-Party Computation

Multi-party computation (MPC) considers how a group of people, or parties, can collaboratively compute some function. Each party P_i has a secret input w_i, the players agree on the function ϕ that takes as input the input from each player. The group wants to compute $x = \phi(w_1, \ldots, w_n)$ such that the correct value of x is computed (*correctness*) and x is the *only* new information that is released (*privacy*).

The *view* of a player is defined to be all the values that the player sees during the execution of the protocol. This includes their private input and the output of the protocol, as well as any values communicated by other players during the evaluation of the circuit.

The security of an MPC protocol is typically described in terms of how many players we can allow to be corrupted before they are able to break privacy. In some protocols, we require an honest majority, that is we need the number of corrupted players t to be less than half of the total number of players, n. Sometimes, however, this assumption is too strong, and in particular it is meaningless in two-party protocols. Thus MPC protocols must also handle the dishonest majority assumption, and sometimes even all-but-one player is dishonest.

We also distinguish between semihonest and malicious players. A semihonest player is what we call honest, but curious. They will follow the protocols instructions, but may try to gain more information from what they see in addition to the output. Malicious players however, may not follow the instructions correctly, but do something different in order to compromise the privacy of the inputs or the correctness of the output. Thus we can distinguish between the security of a protocol in the semihonest model and in the malicious model.

5.1 Secret sharing

Secret sharing is the idea of splitting a secret into several pieces such that having access to only one or a limited number of pieces does not give you any information about the secret itself. We can decide that we want to split the secret into n pieces, and that the secret can be reconstructed if and only if more than t of these shares are combined (for $t < n$). This is then called a (t, n)-threshold secret sharing scheme.

A special case of this that is often wanted is a $(n-1, n)$-threshold scheme. This scheme requires all shares to be combined in order to reconstruct the secret. A simple scheme that satisfies this is introduced in Example 9.

Example 9. **($(n-1, n)$-threshold secret sharing scheme)**
Let G be a finite group.

Share*: To share $s \in G$, for each $i \in \{1, 1, \ldots, n-1\}$, choose a uniformly random $s_i \in G$. Then comptue $s_n = s - (\sum_{i=1}^{n-1} s_i)$.*

Reconstruct*: To reconstruct from shares $s_1, s_2, \ldots, s_n \in G$, compute $s = \sum_{i=1}^{n} s_i$*

Privacy*: Knowing only the first $n-1$ shares means you only know uniformly random values, so clearly this reveals no information about the secret s. If on the other hand you know all the shares except some s_i for $i < n$, you know the share that is actually computed from the secret s. However, this value is determined by all the uniformly random values s_i for $i < n$, so when there is one of those you do not know, since it is uniformly random and independent from the other shares s_i for $i < n$ you still learn no information about the secret s. Therefore, in order to reconstruct, all the n shares are needed.*

5.2 Evaluating circuits

The function we will evaluate must be expressed in terms of an algebraic circuit, as discussed in Section 4.4. This circuit will consist of addition gates and multiplication gates. The MPC protocol is a communication protocol for evaluating the circuit, with instructions for what each player must do at each gate. The input to the function we evaluate is shared (using secret sharing) between all the parties that will participate in the evaluation. Each party will evaluate the circuit on their input, which is their share of the input to the function, and communicate with each other as specified by the protocol. In the end, the evaluation of the function is output from the protocol.

One important aspect of this evaluation is that the computation of different types of gates in the circuit will have different computational and communication cost associated to them. Typically, an addition gate and a multiplication by a constant-gate are very cheap to compute as they require no communication between the players. However, multiplication gates require the players to interact. So the cost of evaluating the circuit is influenced by its multiplicative complexity.

Clearly, if the designer of an MPC protocol can influence the choice of circuit, they can influence the cost of the protocol. In PICNIC (see Chapter 7) we use a block cipher as a one-way function and the proof of knowledge uses multiparty computation (specifically, it uses MPC-in-the-head, see Section 5.3) to evaluate the block cipher. The most obvious choice for a block cipher would be AES, but as it has quite a high multiplicative complexity, this increases the computational cost of the proof of knowledge. To reduce the cost of this computation, the block cipher LowMC (see Section 4.5) is chosen. LowMC was designed for, among other things, MPC protocols and has very low multiplicative complexity (MC).

5.3 MPC-in-the-head

A technique called MPC-in-the-head [12] can be used to turn a secure MPC protocol into a zero-knowledge proof. The technique shows the connection between two fundamental notions: zero-knowledge proofs and secure multiparty computation. We take advantage of the security of an MPC protocol to achieve zero knowledge, by having the prover execute an MPC protocol "in the head".

We have statements $x \in L$ in our language, and witnesses $w \in W$ for these statements. We have a relation R such that $(x, w) \in R$ if and only if w is a witness for x. Let the function $\phi(\cdot)$ realise the relation R:

$$\phi(x, w) = 1 \iff (x, w) \in R.$$

Let Π_ϕ be an n-party MPC protocol (with correctness) that evaluates the function ϕ, with $n \geq 3$. Then Π_ϕ takes as input the statement x, and the n input shares w_i. The input shares w_i are the shares produced by secret sharing the witness w for x:

$$w = \sum_{i=1}^{n} w_i.$$

So we have that

$$\Pi_\phi(x, w_1, \dots, w_n) = \phi(x, w_1 + \dots + w_n),$$

and $\Pi_\phi(x, w_1, \dots, w_n) = 1$ if and only if $(x, w_1 + \dots + w_n) \in R$ (otherwise, the output is 0).

The MPC protocol, Π_ϕ, may involve an arbitrary number of parties, n, and needs to be secure against two semi-honest players. The statement x is known to all players and w_i is the private input to player i. The output is received by all players: when $\Pi_\phi(x, w_1, \dots, w_n) = 1$ each player i also outputs 1. Additionally, let Commit be a secure commitment scheme.

The zero-knowledge proof of knowledge protocol Π_R begins with the prover secret-sharing the witness w (to itself) into n additive shares, $w_1, \dots, w_n$. The prover then runs the n-party MPC protocol Π_f "in the head", using the shares $w_1, \dots w_n$ as input for the n "players". The prover stores the view, $view_i$ of each player i containing their private input, w_i, any values they see during the protocol execution and their output (which is 0 or 1). After this is completed, the prover starts their interaction with the verifier.

The prover commits to the view of each player and sends these commitments and the statement x to the verifier. The verifier picks at random two distinct players i, j and challenges the prover to open the views of these players. The prover sends the view of player i and j to the verifier. The verifier accepts if the opened commitments are the one they asked for, and the views are consistent with each other.

The protocol is formally described in Figure 5.1.

We will show in Theorem 4 that the protocol Π_B we get using this technique is a secure zero knowledge proof. The security of the protocol is analysed in the *commitment-hybrid model*. In this model, all parties have access to an idealised implementation of commitments. We need to prove that the protocol has correctness, soundness and HVZK.

Let x be a statement in our language and w a witness for x: $(x, w) \in R$. Let ϕ be the function that corresponds to the relation R: $\phi(x, w) = 1 \iff (x, w) \in R$.
Let Π_ϕ be the MPC protocol that realises ϕ with $n = 3$ players:
$b \leftarrow \Pi_\phi(x, w_1, w_2, w_3)$, where $b \in \{0, 1\}$ such that Π_ϕ outputs 1 if and only if $\phi(x, w_1 + w_2 + w_3) = 1$. If Π_ϕ outputs 1, then so does each player.
Both the prover and the verifier get the statement x as input. The prover also has access to the witness w.

Commit: The prover does the following:

- Secret-shares the witness: Chooses random w_1, w_2, w_3 such that $w_1 + w_2 + w_3 = w$.
- Runs $\Pi_\phi(x, w_1, w_2, w_3)$ ”in the head” to get the views of each player that the prover is emulating, $(view_1, view_2, view_3)$
- Commits to the view of each player: $c_i \leftarrow \mathsf{Commit}(view_i)$ for $i \in \{1, 2, 3\}$.
- Sends (x, c_1, c_2, c_3) to the verifier.

Challenge: The verifier chooses a challenge $(i, j) \leftarrow \{1, 2, 3\}$, and sends it to the prover.

Response: The prover opens the views for players i and j. The prover sends $(view_i, view_j)$ to the verifier.

Verification: The verifier accepts if and only if

- The prover opened the requested views
- $view_i$ and $view_j$ are consistent with each other. with respect to x, the commitments c_i, c_j, and the protocol Π_ϕ
- Both players i and j outputs 1 (which is determined by their view).

Figure 5.1: MPC-in-the-head protocol, Π_R, with 3 players ($n = 3$)

Intuitively, correctness follows from Π_ϕ being correct. Soundness follows from that when there is no witness for the statement, the protocol will not produce the correct output, and zero knowledge follows from Π_ϕ being secure against two semi-honest players.

Theorem 4. (Security of MPC-in-the-head)
Let R be a relation with corresponding funciton ϕ. Let Π_ϕ be an MPC protocol with $n \geq 3$ players. Π_ϕ realises the function ϕ correctly and is secure against two semihonest players. Let Commit *be a secure commitment scheme. Then Π_B from Figure 5.1 is a secure* HVZK *proof (in the commitment hybrid model).*

Proof. (Theorem 4) **Completeness**: In an honest execution of the protocol, the views of the players have been computed correctly and are therefore always consistent with each other and the protocol. Additionally, the commitments will have been computed correctly from these views, so the views will be consistent with the commitments. Because Π_ϕ correctly evaluates the function ϕ, the output shares will correctly recombine to the statement x. So Π_B is correct.

Soundness: Let the statement $x \notin L$ not be in our language, so we have $(x, w) \notin R$ for all possible w, and thus $\forall w \ \phi(x, w) = 0$. Since Π_ϕ correctly realises ϕ, $\Pi_\phi(x, w_1, \dots w_n) = 0$ for all choices for $w_1, \dots, w_n$. There are two possibilities for the views that the prover committed to in the commitment stage: Either they all output 0 (since the MPC protocol must output 0) or there are two views that are inconsistent with each other (since not all views can be consistent and output 1). If they all output 0, then the verifier rejects, so we are done. In the case that they do not all output 0, then we can calculate the probability that the verifier selects an inconsistent pair to challenge the prover on:

$$Pr[(i,j) \text{ inconsistent}] \geq \frac{1}{\binom{n}{2}}$$

Note that we open 2 views because the MPC protocol is secure against 2 semi-honest players.

Zero knowledge: Let $\mathcal{S}_{\text{MPC}}$ be the simulator for the MPC protocol Π_ϕ. We know that such a simulator exists because of the security of Π_ϕ against 2 semi-honest players. We construct a simulator $\mathcal{S}$. Let V^* be a malicious verifier. We run the simulator $\mathcal{S}$ on input x:

1. Run V^* on input x. Let (i, j) be the pair of indices chosen by V^* as a challenge.
2. Simulate the two views $view_i, view_j$ by picking random input shares w_i, w_j and running the MPC simulator $\mathcal{S}_{\text{MPC}}(\{i, j\}, x, (w_i, w_j))$.

Let $(x, w) \in R$, that is $\phi(x, w) = 1$. The randomness of V^* that the simulation outputs is identically distributed to the randomness of V^* in an actual execution. So we just need to show that for any choice, the simulation is perfect.

Let (i, j) be the verifiers challenge. In the real execution, the choice of $w_1, \dots w_n$ are uniform and independent. So the choice made by the simulator is identically distributed. The distribution of the simulated views $(view_i, view_j)$ is distributed identically to the real views for any choice of i and j since the MPC simulator $\mathcal{S}_{\text{MPC}}$ is a perfect 2-private simulator for Π_ϕ. □

Chapter 6

Zero Knowledge for Algebraic Circuits

In this chapter we will introduce the sigma protocol, ZKBoo, that is the basis of the sigma protocol used in PICNIC. ZKBoo is a zero knowledge proof system. In particular, the protocol is used to prove knowledge of some pre-image of a (one-way) function.

ZKBoo makes use of several concepts that we have already seen and will see in this chapter:

- Commitment schemes (Section 2.3)
- MPC, in particular MPC-in-the-head (Section 5.3), including secret sharing (Section 5.1).
- Function decomposition (Section 6.1)

The protocol is modified somewhat when it is used in the PICNIC scheme. We will see these modifications and then the resulting protocol that is called ZKB++ in Section 6.3.

6.1 Function decomposition

The ZKBoo protocol assumes the existence of what we call (2,3)-decompositions of functions. The purpose of this is to split the evaluation of a function on some input into 3 parts. The three parts are carried out by three different players, such that when we recombine the results each of the players obtained, we get the same result we would have obtained if we evaluated the original function on the same input. In addition, we require 2-privacy, which means that revealing 2 (out of 3) views reveals no information about the input.

Let ϕ be the function we want to decompose, w is the input and x is the output such that $x = \phi(w)$. Each player i will evaluate their own component of the function, which we call ϕ_i. The evaluation of the function may consist of several stages, in between which the players interact. The function evaluation can therefore be split into N parts, one for each of these N stages. So each player i for $i \in \{1,2,3\}$ will evaluate all the functions $\cup_{j=1}^{N} \phi_i^{(j)}$. Each of these functions take input from the player itself, player i and the next one, player $i+1$ (where $3+1=1$):

$$view_i[j+1] \leftarrow \phi_i^j(view_i[j], r_i, view_{i+1}[j], r_{i+1}).$$

To simplify notation, we give the name Update to all of the functions $\cup_{j=1}^{N}\{\phi_1^{(j)}, \phi_2^{(j)}, \phi_3^{(j)}\}$. The Update function updates the view of party i:

$$view_i[j+1] \leftarrow \mathsf{Update}(view_i[j], \dots).$$

The input value for each player as well as any intermediate values are stored in their respective views, $view_i$.

The decomposition consists of a tuple of algorithms (Share, Update, Output$_i$, Rec) for $i \in \{1,2,3\}$. The 2-3-decomposition is defined in Defintion 14.

Definition 14. ((2,3)-decomposition)
*A **(2,3)-decomposition** for the function ϕ is a tuple of algorithms*

$$\mathcal{D}_\phi = \{\mathsf{Share}, \mathsf{Update}, \mathsf{Output}_1, \mathsf{Output}_2, \mathsf{Output}_3, \mathsf{Rec}\}$$

such that:

- Share*: Splits the input into three input shares,*

  $$w_1, w_2, w_3 \leftarrow \mathsf{Share}(w; r_1, r_2, r_3).$$

  The function is surjective and potentially randomized.

- Update*: Each player i evaluates* Update *to update their view, $view_i$:*

  $$view_i[j+1] \leftarrow \mathsf{Update}(view_i[j], \dots)$$

- $\forall i \in [3]$: Output$_i$. *The output algorithm for each player i takes in their view, $view_i$ and outputs their output share, x_i.*

- Rec*: This algorithm recombines the output shares into the final output:*

  $$x \leftarrow \mathsf{Rec}(x_1, x_2, x_3),$$

  such that $x = \phi(w)$.

- *Correctness: $\forall x \in L : Pr[\phi(x) = \Pi_\phi^*(x)] = 1$.*

- *2-Privacy: $\forall e \in [3]$: $\exists \mathcal{S}$ (an efficient simulator) such that $\forall x \in L$, $(\{r_i, view_i\}_{i \in \{e,e+1\}}, y_{e+2})$ and $\mathcal{S}_e(\phi, y)$ have the same probability distribution.*

Figure 6.1 shows a visual representation of what the (2,3)-decomposition does.

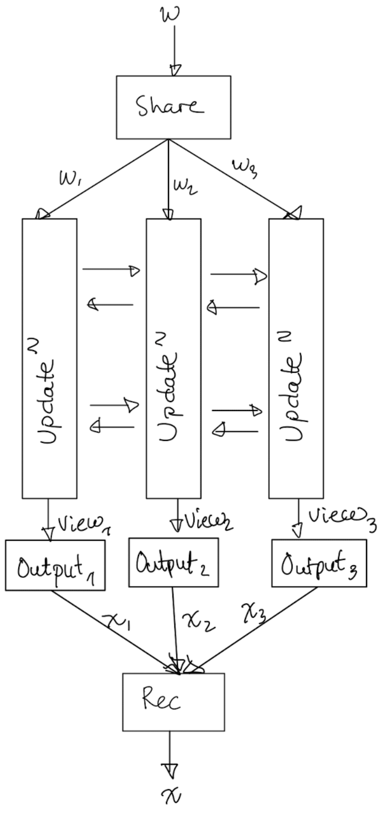


Figure 6.1: Visual representation of a (2-3)-decomposition [10].

In the signature scheme PICNIC, we have a (2,3)-decomposition of the one-way function F constructed from the block cipher LowMC. We consider the LowMC plaintext-ciphertext pair (a, b) as the public key and the LowMC key is the secret key. So we want to prove knowledge of the pre-image of $\mathsf{F}_a(k) = b$, which is the LowMC key, k.

This idea of a (2,3)-decomposition is a generalisation of the MPC-in-the-head technique (see Section 5.3). The decomposition replaces the MPC protocol that is used as a black box in the MPC-in-the-head protocol.

6.2 ZKBoo

The ZKBoo protocol [10] is a sigma protocol that turns the notion of a $(2, 3)$-decomposition (Definition 14) of a function, ϕ, into a zero knowledge protocol. Let the language be $L_\phi = \{x \mid \exists w : \phi(w) = x\}$. Let $x \in L_\phi$ be a statement in our language and let $w \in W$ be a witness for x. The aim is for the prover to convince the verifier that it knows w such that $\phi(w) = x$, but without revealing any information about the witness w. To do this, ZKBoo uses the (2,3)-decomposition construction from Section 6.1, which is based on the MPC-in-the-head technique [12], from Section 5.3.

Both the prover and the verifier get as input the statement $x \in L_\phi$. The prover also knows a witness w for x, so that $x = \phi(w)$, and wants to convince the verifier of this. We are also given a (2,3)-decomposition, Π_ϕ^*, of the function ϕ. The prover will use Π_ϕ^* to do MPC-in-the-head:

- The prover samples some randomness k_1, k_2, k_3, for each of the three "players" in the MPC protocol.
- The input (the witness for our statement) w is shared into three input shares, w_1, w_2, w_3, for each of the three "players". This uses the Share function that Π_ϕ^* specifies.
- The prover will evaluate the decomposed function ϕ according to Π_ϕ^* for each of the three *"players"*. From this, the prover gets the view $view_i$ and the output share x_i for each player $i \in [3]$.
- The view $view_i$ contain everything seen by player i, its input share, output share and any intermediate values and "messages" sent between the players when executing the MPC protocol. If we recombine the output shares x_1, x_2, x_3 using the function $\mathsf{Rec}(x_1, x_2, x_3)$ we will get the output $x = \phi(w)$.

After obtaining the views for the players and their output shares, the prover commits to each view $view_i$ using a Commit function (see Section 2.3) that takes in the view, $view_i$ and randomness of player i, k_i. The prover sends the following to the verifier: $a = (x_1, x_2, x_3, c_1, c_2, c_3)$.

Now the verifier gets to choose a challenge for the prover. The verifier chooses some index $e \in [3]$, challenging the prover to open the commitments c_e and c_{e+1}.

The prover responds with the required openings, sending $z = (view_e, view_{e+1}, r_e, r_{e+1})$ to the verifier.

In the end, the verifier makes some checks, to make sure everything was computed as it should have been. If any of the checks fail, the verifier outputs Reject. If they all succeed, the verifier outputs Accept. The checks are:

- Check that the output shares recombine to the input value: $x \stackrel{?}{=} \mathsf{Rec}(x_1, x_2, x_3)$.
- Check that the output shares are consistent with the view:
  $\forall i \in \{e, e+1\} : x_i \stackrel{?}{=} \mathsf{Output}(view_i)$.
- Now we consider ϕ_i to be the composition of N distinct functions such that $\phi_i = \phi_i^N \circ \phi_i^{N-1} \circ \cdots \circ \phi_i^1$. Denote the view of player e at the stage immediately after evaluating ϕ_e^j to be $view_e[j]$ (thus $view_e[N] = view_e$, and $view_e[0] = w_e$).

  Check that the view at any specific stage, $j \in [N]$, $view_e[j]$ is equal to the output of that particular function with the openings as input: $view_e[j] \stackrel{?}{=} \phi_e^j(view_e, view_{e+1}, r_e, r_{e+1})$.

In Theorem 5 we prove the security of ZKBoo. Figure 6.2 shows the ZKBoo protocol.

Input:

- The verifier and prover both have input $x \in L_\phi$
- The prover knows w such that $x = \phi(w)$.
- A (2,3)-decomposition, Π^*_ϕ, is given

Commit: The prover does:

1. Sample random tapes r_1, r_2, r_3
2. Run $\Pi^*_\phi(x)$ and obtain the views $view_1, view_2, view_3$ and the output shares x_1, x_2, x_3
3. Commit to the views: $\forall i \in [3] : c_i \leftarrow \mathsf{Commit}(r_i, view_i)$
4. Send to the verifier: $a = (x_1, x_2, x_3, c_1, c_2, c_3)$

Prove:

1. Verifier: chooses index $e \in [3]$ and sends it to the prover
2. Prover: Answers the verifiers challenge by opening the commitments c_e, c_{e+1}: Sends to the verifier $z = (r_e, view_e, r_{e+1}, view_{e+1})$.

Verify: The verifier does:

1. If $\mathsf{Rec}(x_1, x_2, x_3) \neq y$, output Reject
2. If $\exists i \in \{e, e+1\}$ such that $x_i \neq \mathsf{Output}_i(view_i)$, output Reject
3. If $\exists j$ such that $view_e[j] \neq \phi_e^{(j)}(view_e, view_{e+1}, r_e, r_{e+1})$, output Reject
4. Output Accept

Figure 6.2: The ZKBoo protocol [10].

Theorem 5. (ZKBoo)
The sigma protocol ZKBoo *in Figure 6.2 has 3-special soundness and is honest verifier zero knowledge.*

Proof. (Theorem 5) ZKBoo has the sigma protocol communication pattern and completeness is given by the correctness of the (2,3)-decomposition of ϕ, Π^*_ϕ.

3-special soundness: Consider three accepting conversations $(a, i, z_i), i \in [3]$:

- $(a, 1, z_1)$: $z_1 = (r_1, view_1, r_2, view_2)$,
- $(a, 2, z_2)$: $z_2 = (r_2, view_2, r_3, view_3)$,
- $(a, 3, z_3)$: $z_3 = (r_3, view_3, r_1, view_1)$.

Note that $view_1 \in z_1 = view_1 \in z_3$, and the same holds for $view_2$ and $view_3$. This is guaranteed by the binding property of the commitment.

Now we move backwards through the decomposition: Since the three conversations are accepting we have $\forall i : y_i = \mathsf{Output}(view_i)$ and all the entires in $view_i$ are correct. So since Share is surjective, we can find $w' = \mathsf{Rec}(view_1[0], view_2[0], view_3[0]) = \mathsf{Rec}(w_1, w_2, w_3)$.

Note that with 2 accepting conversations, even though all input shares are included (so one would think this was enough to extract the witness), one branch of computation is not checked, so there may be the case that $\exists i : w_1 \neq \phi_i^{(j)}(view_i, view_{i+1}, r_i, r_{i+1})$. So ZKBoo does not satisfy 2-special soundness.

HVZK: We construct a simulator $\mathcal{S}$.

1. Input: $x \in L_\phi$ and $e \in [3]$.
2. Run 2-privacy simulator (which must exist due to the 2-privacy property of Π^*_ϕ). This returns $(\{r_i, view_i\}_{i \in \{e,e+1\}}, x_{e+2})$
3. $\mathcal{S}$ sets $view_{e+2} = 0^{|view|}$, $r_{e+2} = 0^{|r|}$.
4. $\mathcal{S}$ constructs a commitment to the three views.

The simulator $\mathcal{S}$ perfectly simulates the proofs. The adversary cannot distinguish between real proofs and simulated proofs, so the protocol is HVZK.

It now follows from Theorem 1 that ZKBoo is a proof of knowledge, and thus a zero knowledge proof of knowledge. □

6.3 ZKB++

In the construction of PICNIC, several alterations are made to ZKBoo as it is presented in Section 6.2. The resulting sigma protocol is called ZKB++ [5]. The changes made result in a reduction of the transcript size by more than half, without increasing the computation cost. The changes are presented as six different optimisations:

Optimisation 1 - the Share function: The Share function samples shares pseudorandomly:

$$(w_1, w_2, w_3) \leftarrow \mathsf{Share}(w, r_1, r_2, r_3) \coloneqq w_1 = R_1(0), w_2 = R_2(0), w_3 = w - w_1 - w_2$$

Where R_i is a pseudorandom generator with seed r_i. Note that 2-privacy is preserved. This enables the verifier to compute the input shares to players 1 and 2 using their random tape as seed. However, the verifier cannot compute the input share to player 3.

Optimisation 2 - not including input shares: Due to optimisation 1, the prover can omit the input shares that the verifier can compute from its messages. If the challenge from the verifier is $e = 1$, then the input shares can be omitted from the response. If $e = 2$ or $e = 3$, then one input share must be sent since w_3 cannot be computed by the verifier. Since the challenge is uniformly random, the expected number of input shares that needs to be included by the prover is $2/3$.

Optimisation 3 - not including commitments: It is not necessary to include the commitments to all three views because for the two opened views, the verifier can (and will anyway) recompute the commitment. So the commitment must ble explicitly included in the message only for the view that is not opened. Note that the challenge e is computed from all the commitments (and some other values), so the challenge functions as a commitment to the commitments. Hence there is no loss of security by doing this.

Optimisation 4 - no additional randomness for commitments: r_i (the seed value) is the first input to the commitment. So the protocol input to the commitment doubles as a randomization value. Since r_i is used both as the seed for the PRG and as randomness for the commitment, we need to work in the random oracle model - which we already do in order to make the proofs non-interactive, so there is no security loss here.

Optimisation 5 - not including the output shares: In ZKBoo the output shares are included in the value a (the commitment phase). And later, two of the output shares are included when two views are opened. Clearly, it is not needed to send these two times. However, it is not necessary to send the output shares at all, because the verifier can recompute them from the rest of the proof. For the two views that are opened, the ouput share is the value on the output wire, so it is easily computed since the verifier has access to the random tapes and any communicated bits. For the third output value, this can be computed from the input, x and the two other input shares x_e, x_{e+1} since this is how it was computed in the first place.

Optimisation 6 - not including $view_e$: The verifier recomputes every wire in $view_e$ in its verification to check that the values they received were correct. However, we can omit sending these values as the verifier can compute them given just the random tapes r_e, r_{e+1} and the values of $view_{e+1}$. Then the verifier uses this recomputed view to check the commitments, which will only verify if everything was computed correctly due to the binding property of the commitment.

A detailed analysis of the computational cost of this is included in [5].

In addition to this, the ZKB++ protocol also applies a non-interactive transform. Figure 6.3 shows the prover algorithm of the ZKB++ protocol and figure 6.4 shows the verifier algorithm. In the figures, the Fiat-Shamir transform is used to make the protocol non-interactive. In PICNIC, ZKB++ is made non-interactive using Unruh's transform, which is secure in QROM. This application of Unruh's transform is also specialised to reduce the overhead to only 1.6x compared to 4x for a direct application.

Input: The function ϕ and statement $x \in L_\phi$ is public. The prover knows w such that $x = \phi(w)$. Hash functions H, H', G used by both parties, modelled as random oracles. t is the number of parallel iterations.

Prover: $\pi \leftarrow \mathsf{Prove}(x, w)$

1. For each $s_i : i \in [1, t]$: Sample random tapes $r_1^{(i)}, r_2^{(i)}, r_3^{(i)}$. Simulate MPC protocol, for each player j compute:

$$\begin{aligned}(w_1^{(i)}, w_2^{(i)}, w_3^{(i)}) &\leftarrow \mathsf{Share}(w, r_1^{(i)}, r_2^{(i)}, r_3^{(i)}) \\ &= (\mathsf{G}(r_1^{(i)}, \mathsf{G}(r_2^{(i)}), w \oplus \mathsf{G}(r_1^{(i))} \oplus \mathsf{G}(r_2^{(i)}), \\ view_j^{(i)} &\leftarrow \mathsf{Update}(\ldots \mathsf{Update}(w_j^{(i)}, w_{j+1}^{(i)}, r_j^{(i)}, r_{j+1}^{(i)}) \ldots), \\ x_j^{(i)} &\leftarrow \mathsf{Output}(view_j^{(i)}).\end{aligned}$$

   Commit $[C_j^{(i)}, D_j^{(i)}] \leftarrow [\mathsf{H}'(r_j^{(i)}, view_j^{(i)}), r_j^{(i)} \parallel view_j^{(i)}]$.

   Let $a^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, C_1^{(i)}, C_2^{(i)}, C_3^{(i)})$.

2. Compute challenge: $e \leftarrow \mathsf{H}(a^1, \ldots a^t)$. Get $e_i \in \{1, 2, 3\}$ for $i \in [1, t]$.

3. For each $s_i : i \in [1, t]$: let $b^i = (y_{e^i+2}^{(i)}, C_{e^i+2}^{(i)})$. Set:

$$z^{(i)} \leftarrow \begin{cases}(view_2^{(i)}, r_1^{(i)}, r_2^{(i)}) & \text{if } e^{(i)} = 1, \\ (view_3^{(i)}, r_2^{(i)}, r_3^{(i)}, w_3^{(i)}) & \text{if } e^{(i)} = 1, \\ (view_1^{(i)}, r_3^{(i)}, r_1^{(i)}, w_3^{(i)}) & \text{if } e^{(i)} = 3.\end{cases}$$

4. Output $\pi \leftarrow [e, (b^{(1)}, z^{(1)}), (b^{(2)}, z^{(2)}), \ldots, (b^{(t)}, z^{(t)})]$.

Figure 6.3: The ZKB++ protocol [5] prover algorithm

Input:

- The function ϕ and statement $x \in L_\phi$ is public.
- Hash functions H, H', G used by both parties, modelled as random oracles
- t is the number of parallel iterations

Verifier: $b \leftarrow \mathsf{Verify}(x, \pi)$ where $b \in \{\mathsf{Accept}, \mathsf{Reject}\}$

1. For each $s_i : i \in [1, t]$: Run the MPC protocol to reconstruct the views, input and output shares that were not given as part of the proof π.

$$w^{(i)}_{e^{(i)}} \leftarrow \begin{cases} \mathsf{G}(r^{(i)}_1 & \text{if } e^{(i)} = 1, \\ \mathsf{G}(r^{(i)}_2 & \text{if } e^{(i)} = 1, \\ w^{(i)}_3 \text{ from } z^{(i)} & \text{if } e^{(i)} = 3. \end{cases} x^{(i)}_{e^{(i)}+1} \leftarrow \begin{cases} \mathsf{G}(r^{(i)}_2) & \text{if } e^{(i)} = 1, \\ w^{(i)}_3 \text{from } z^{(i)} & \text{if } e^{(i)} = 2, \\ G(k^{(i)}_1) & \text{if } e^{(i)} = 3. \end{cases}$$

Get $view^{(i)}_{e^{(i)}+1}$ from $z^{(i)}$. Compute:

$$\begin{aligned} view^{(i)}_e &\leftarrow \mathsf{Update}(\ldots \mathsf{Update}(w^{(i)}_e, w^{(i)}_{e+1}, r^{(i)}_e, r^{(i)}_{e+1}) \ldots), \\ x^{(i)}_{e^{(i)}} &\leftarrow \mathsf{Output}(view^{(i)}_{e^{(i)}}), \\ x^{(i)}_{e^{(i)}+1} &\leftarrow \mathsf{Output}(view^{(i)}_{e^{(i)}+1}), \\ x^{(i)}_{e^{(i)}+2} &\leftarrow x \oplus x^{(i)}_{e^{(i)}} \oplus x^{(i)}_{e^{(i)}+1} \end{aligned}$$

Compute commitments for the views. For $j \in \{e^{(i)}, e^{(i)} + 1\}$:

$$[C^{(i)}_j, D^{(i)}_j] \leftarrow [\mathsf{H}'(k^{(i)}_j, view^{(i)}_j), r^{(i)}_j \| view^{(i)}_j]$$

Let $a'^{(i)} = (x^{(i)}_1, x^{(i)}_2, x^{(i)}_3, C^{(i)}_1, C^{(i)}_2, C^{(i)}_3)$.
Note that $y^{(i)}_{e^{(i)}+2}$ and $C^{(i)}_{e^{(i)}+2}$ is part of $z^{(i)}$.

2. Compute the challenge: $e' = \mathsf{H}(y^{(i)}_1, y^{(i)}_2, y^{(i)}_3, C^{(i)}_1, C^{(i)}_2, C^{(i)}_3)$.
If $e = e'$, output Accept, otherwise output Reject.

Figure 6.4: The ZKB++ protocol [5] verifier algorithm

Theorem 6. (Security of the ZKB++ NIZK proof of knowledge protocol)
The non-interactive scheme ZKB++ *from Figures 6.3 and 6.4 is a* HVZK *proof of knowledge with simulation-sound online extractability.*

Proof sketch. (Theorem 6) The result follows from the security of ZKBoo (Theorem 5) and that Unruh's transform preserves this security (Theorem 3). The optimisations made to ZKBoo in order to construct ZKB++ do not alter the security properties, so the result follows. The full proof can be found in the paper introducing the PICNIC signature scheme [5]. □

Chapter 7

The PICNIC Signature Scheme

We will now describe the PICNIC signature scheme [5]. The purpose of PICNIC is to provide a post-quantum secure signature scheme. PICNIC is an instantiation of the general signature scheme construction showed in Example 4.

Over the past few chapters we have described the building blocks that are used in PICNIC. We will now put them together and show that this yields a secure signature scheme.

We will see that the security of PICNIC reduces entirely to the security of the building blocks that are used. Since these building blocks rely only on the security of symmetric-key primitives that are considered post-quantum secure, this shows that PICNIC is indeed post-quantum secure.

7.1 Algorithm overview

The PICNIC signature scheme consists of three algorithms KeyGen, Sign and Verify. Figure 7.1 shows how the algorithms interact, and Figure 7.2 shows what each algorithm does.

We will discuss each algorithm separately, including how it works, what primitives and sub-protocols are chosen and what security we require.

Key generation algorithm, $\mathsf{KeyGen}()$**:** The key generation algorithm uses a keyed one-way function F. A random secret key is sampled: $k_s \xleftarrow{\$} \{0,1\}^n$. Also, a random value is sampled: $r_{\mathsf{KeyGen}} \xleftarrow{\$} \{0,1\}^n$. The public key is computed as $k_p \leftarrow \mathsf{F}_{r_{\mathsf{KeyGen}}}(k_s)$, that is the evaluation of the function F with key r_{KeyGen} in the value k_s. The algorithm outputs a key pair: $(sk = (r_{\mathsf{KeyGen}}, k_s, k_p), pk = (r_{\mathsf{KeyGen}}, k_p))$.

We require that the function F is a one-way function such that knowing r_{KeyGen} and k_p does not reveal (any information about) the pre-image k_s.

In PICNIC, F is instantiated with LowMC [1]. LowMC is introduced in Section 4.5 and is a block cipher that has particularly low multiplicative complexity, which is beneficial

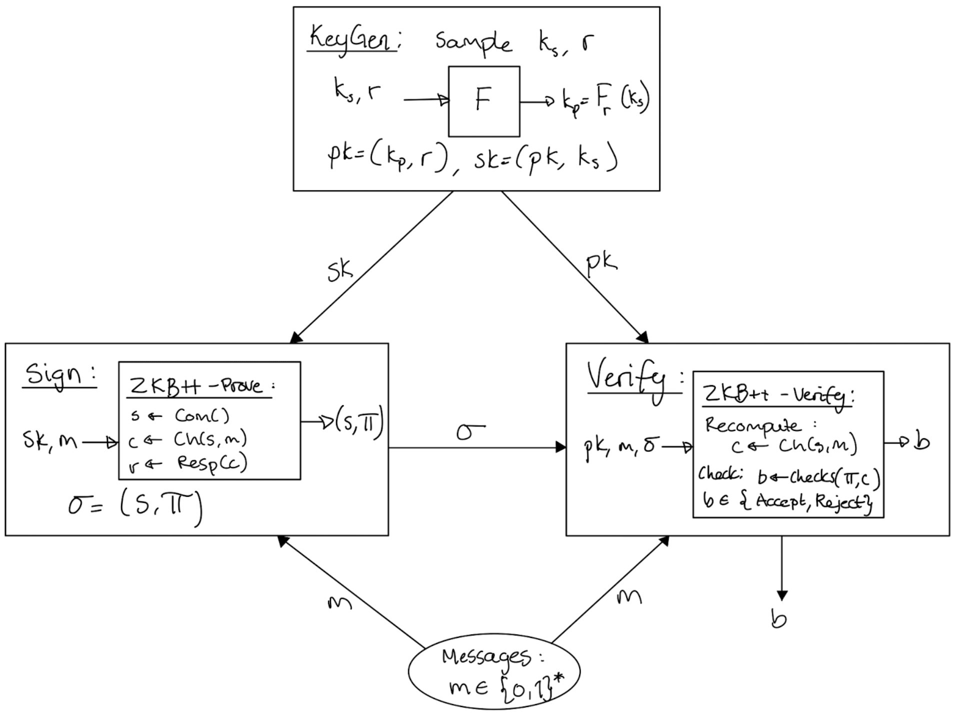


Figure 7.1: The PICNIC signature scheme with its three algorithms: KeyGen, Sign, and Verify.

for the computational cost of the protocol as discussed previously. This is done such that the preimage of the one-way function F is the LowMC key. Let $\mathcal{K}$ be the set of LowMC keys, and $\mathcal{S}$ the set of messages (plaintexts and ciphertexts). Let $r_{\mathsf{KeyGen}} \in \mathcal{S}$ be the key for F. Then we have:

$$\begin{aligned} \mathsf{F}_{r_{\mathsf{KeyGen}}} &: \mathcal{K} \to \mathcal{S}, \\ & k_s \mapsto \mathsf{Enc}_{k_s}(r_{\mathsf{KeyGen}}) = k_p, \end{aligned}$$

where $\mathsf{Enc}_{k_s}(r_{\mathsf{KeyGen}})$ is the LowMC encryption of r_{KeyGen} under the key k_s.

Signing algorithm, Sign **(m, sk):** The signing algorithm takes a message to be signed, and the secret key as input. The signature consists of a NIZK proof of knowledge of the secret key k_s that depends on the message m.

The NIZK prover algorithm $\mathsf{Prove}_\mathsf{H}$ consists of three phases: the commit phase, the challenge phase and the response phase. The commitment phase outputs a value $s \leftarrow \mathsf{Com}()$ that is input to the challenge phase. The challenge phase takes this value s and the message m and produces the challenge as $c = \mathsf{H}(s, m) \leftarrow \mathsf{Ch}(s, m)$. The response phase outputs the proof of knowledge for k_s, $\pi \leftarrow \mathsf{Resp}(c)$:

$$\pi \leftarrow \mathsf{Prove}_\mathsf{H}(m, s, sk).$$

```
**KeyGen** ( ):
    $k_s \xleftarrow{\$} \{0,1\}^n, r_{\mathsf{KeyGen}} \xleftarrow{\$} \{0,1\}^n$
    $k_p \leftarrow \mathsf{F}_{k_s}(r_{\mathsf{KeyGen}})$
**return** $(sk = (r_{\mathsf{KeyGen}}, k_s, k_p), pk = (r_{\mathsf{KeyGen}}, k_p))$

**Sign** $(sk, m)$:
    Compute $\sigma = (s, \pi) \leftarrow \mathsf{Prove}_{\mathsf{H}}((r_{\mathsf{KeyGen}}, k_p, m), k_s)$
    The challenge used in $\mathsf{Prove}_{\mathsf{H}}$ is computed according to Unruh's transform,
     $c \leftarrow \mathsf{H}(s, m)$.
**return** $\sigma$

**Verify** $(pk, m, \sigma)$:
    $b \leftarrow \mathsf{Verify}_{\mathsf{H}}((r_{\mathsf{KeyGen}}, k_p, m), \pi), b \in \{\mathsf{Accept}, \mathsf{Reject}\}$
**return** $b$
```

Figure 7.2: Generic description of the PICNIC algorithm. This is instantiated with ZKB++, which is made non-interactive using Unruh's transform, as $(\mathsf{Prove}_{\mathsf{H}}, \mathsf{Verify}_{\mathsf{H}})$ and a secure pseudorandom function. LowMC will act as this secure PRF.

The signing algorithm outputs the commitment value s and the proof π:

$$\sigma = (s, \pi) = \mathsf{Sign}(m, sk).$$

The prover sends the signature σ to the verifier.

In PICNIC, $\mathsf{Prove}_{\mathsf{H}}$ (and the verification algorithm that belongs to it, $\mathsf{Verify}_{\mathsf{H}}$) is instantiated with the sigma protocol ZKB++, which is made non-interactive using Unruh's transform (see Section 3.4.2). ZKB++ is adapted from ZKBoo as described in Section 6.3. Unruh's transform gives us a NIZK proof system with HVZK that is simulation-sound online-extractable in QROM when applied to a sigma protocol that is a proof of knowledge with completeness, HVZK, and n-special soundness. For the resulting NIZK protocol to be secure, we require that the sigma protocol satisfies these conditions. As shown in Chapter 6, ZKB++ has 3-special soundness and completeness, and is HVZK.

For the signature we require EUF-CMA security. This means that even if the adversary has seen pairs of messages and signatures (m, σ), they should not be able to produce a valid signature σ' on a new message $m' \neq m$. We will see the security of PICNIC in Section 7.2.

In the introduction of ZKB++ in Chapter 6, we do not show the protocol made non-interactive with Unruh's transform. The Fiat-Shamir transform is used in the algorithm descriptions to illustrate non-interactivity. However, as discussed in Chapter 3, we do not consider the Fiat-Shamir transform to be secure in QROM. Thus, we use Unruh's transform instead for PICNIC since we require security against quantum adversaries.

Using Unruh's transform gives us a large overhead in the proof size, which are roughly four times as large as when using the Fiat-Shamir transform. However, PICNIC introduces some optimisations to reduce this overhead so that it becomes only approximately 1.6 times the Fiat-Shamir transform proofs, making the proof size acceptably small.

Verification algorithm, Verify **(**σ**,** m**,** pk**):** The verification algorithm takes a signature, a message and the public key as input. The verifier runs the verification algorithm $\mathsf{Verify_H}$. This is the verification algorithm that is part of the NIZK proof system $\Pi_H = (\mathsf{Prove_H}, \mathsf{Verify_H})$. This proof system is the ZKB++ protocol made non-interactive with Unruh's transform as described above. The verification algorithm outputs b such that:

$$b \leftarrow \mathsf{Verify_H}(m, \sigma, pk) : b \in \{\mathsf{Accept}, \mathsf{Reject}\}.$$

We require that the Verify algorithm outputs $b = \mathsf{Accept}$ if and only if for $\sigma = (s, \pi)$, π is a valid proof of knowledge of sk with challenge $c \leftarrow \mathsf{H}(s, m)$ (that is, the proof is correct and sound).

7.2 Security of Picnic

We are working in the QROM security model. The security model is introduced in Section 2.4.1.

We will prove that the PICNIC security scheme is EUF-CMA secure. We do this by reducing the security of the signature scheme to the security of the building blocks that we have already discussed and proven to be secure. So the security proof holds for all signature schemes of the same construction, when secure building blocks are chosen.

For PICNIC in particular, we rely on the one-wayness of the function we use based on LowMC, and the security of the NIZK proof of knowledge protocol ZKB++ that is made non-interactive using Unruh's transform, which preserves the security of the interactive zero knowledge proof of knowledge.

Theorem 7. (EUF-CMA **of PICNIC)**

Let PICNIC *be the signature scheme from Figure 7.2 instantiated with* ZKB++ *as* $(\mathsf{Prove_H}, \mathsf{Verify_H})$ *and a one-way function,* F. *Let $\mathcal{A}$ be a (q, t)-adversary against the* EUF-CMA *security of* PICNIC *in the* QROM *model that makes at most q queries to the experiment and the runtime of $\mathcal{A}$ and the experiment is at most t. Then there is an adversary $\mathcal{B}$ against the security of our one-way function* F*, with runtime approximately t, such that:*

$$Adv^{\mathsf{EUF\text{-}CMA}}_{\text{PICNIC}}(\mathcal{A}) = Adv^{OW\text{-}sec}_{\mathsf{F}}(\mathcal{B})$$

Proof. (Theorem 7) First, we note that PICNIC is correct, which follows from the completeness of the proof system ZKB++.
We proceed with a sequence of games:

Game 1: This is the original EUF-CMA security experiment. The prover responds to the adversary $\mathcal{A}$ with signatures $\sigma = (s, \pi)$. We get

$$Adv^{\mathsf{EUF\text{-}CMA}}_{\text{PICNIC}}(\mathcal{A}) = Pr[G_1 = 1].$$

Game 2: Now the prover will not use $\mathsf{Prove_H}$ to create genuine proofs π as part of the signature $\sigma = (s, \pi)$. Instead, the prover will simulate the proofs using the simulator $\mathcal{S}_\mathsf{H}$ for the $(\mathsf{Prove_H}, \mathsf{Verify_H})$ proof system and the simulator $\mathcal{S}_{\text{init}}$ to choose the oracles H and G, as presented in Section 3.4.2.

Since the simulator $\mathcal{S}_\mathsf{H}$ perfectly simulates the zero knowledge proofs, and the simulator $\mathcal{S}_{\text{init}}$ chooses random oracles in such a way that the adversary cannot distinguish between the chosen oracles and the random oracles, the adversary $\mathcal{A}$ cannot distinguish between Game 1 and Game 2. Therefore we have

$$Pr[G_1 = 1] = Pr[G_2 = 1].$$

Note that the prover does not use the secret key sk to create proofs now. This effectively removes the signature queries from the experiment, because the adversary cannot learn anything from them, since the secret key was not involved in creating the simulated signatures. In fact, the adversary would be able to do the simulation themselves.

Suppose the adversary $\mathcal{A}$ is able to produce a forgery σ for message $m.$ Since the signature is a proof of knowledge, that means we have an extractor $\mathcal{E}_{\Pi}$ that will extract a witness from the proof of knowledge (except with negligible probability). This extractor is the extractor $\mathcal{E}_{\Pi_U}$ from Section 3.4.2.

The witness that is extracted by $\mathcal{E}_{\Pi}$ is the preimage of our one-way function F. So we can build an adversary $\mathcal{B}$ against the one-wayness (OW-sec) of our function F. $\mathcal{B}$ uses the adversary $\mathcal{A}$ and the extractor $\mathcal{E}_{\Pi}$ to find a preimage of the one-way function F. The adversary $\mathcal{B}$ finds a preimage of F whenever $\mathcal{A}$ successfully forges a signature, by using the extractor, and spends approximately the same time as $\mathcal{A}$. Since $\mathcal{B}$ succeeds whenever $\mathcal{A}$ succeeds, we have

$$Adv_{\mathsf{F}}^{\text{OW-sec}}(\mathcal{B}) = Pr[G_2 = 1].$$

The result follows. $\square$

Chapter 8

Conclusions

We have seen how the PICNIC signature scheme is constructed and proved its security against EUF-CMA attacks. This security result applies to other signature schemes that uses the same construction, such as the schemes mentioned in Section 1.1.2, BBQ and BANQUET.

We have seen that the security of the scheme reduces to symmetric-key primitives, and thus it is secure against quantum adversaries, without any number theoretic hardness assumptions and also without any hardness assumptions on other public key structures. Because of this, the construction is interesting, even if it is rather complicated.

The signatures made with PICNIC, or one of its friends, are rather large, but with further work, it is possible that this can be reduced to make the scheme more practical.

As of June 2021 the NIST standardisation process is still in round 3, evaluating finalists. As an alternate digital signature scheme, PICNIC is not a candidate for standardisation now, but it may be considered in the future after further study.

Bibliography

[1] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056, pages 430–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[2] Carsten Baum, Cyprien Delpech de Saint Guilhem, Daniel Kales, Emmanuela Orsini, Peter Scholl, and Greg Zaverucha. Banquet: Short and Fast Signatures from AES. Technical Report 068, 2021.

[3] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random Oracles in a Quantum World. Technical Report 428, 2010.

[4] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. Self-published, web, 0.5 edition.

[5] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. *Cryptology ePrint Archive, Report 2017*, 279, March 2017.

[6] Information Technology Laboratory Computer Security Division and National Institute of Standards and Technology (NIST). Public-Key Post-Quantum Cryptographic Algorithms: Nominations | CSRC, December 2016.

[7] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, Cambridge, 2015.

[8] Cyprien Delpech de Saint Guilhem, Lauren De Meyer, Emmanuela Orsini, and Nigel P. Smart. BBQ: Using AES in Picnic Signatures. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, Lecture Notes in Computer Science, pages 669–692, Cham, 2020. Springer International Publishing.

[9] Hanwen Feng, Jianwei Liu, and Qianhong Wu. Secure Stern Signatures in Quantum Random Oracle Model. In Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis, editors, *Information Security*, volume 11723, pages 425–444. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.

[10] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. Cryptology ePrint Archive, Report 2016/163, 2016. `https://eprint.iacr.org/2016/163`.

[11] Shafi Goldwasser and Mihir Bellare. Lecture Notes on Cryptography. *2008*, page 289.

[12] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM Journal on Computing*, 39(3):1121–1152, 2009.

[13] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, November 2014. Google-Books-ID: OWZYBQAAQBAJ.

[14] Wenbo Mao. *Modern Cryptography: Theory and Practice*. Prentice Hall PTR, 2004. Google-Books-ID: H42WQgAACAAJ.

[15] Information Technology Laboratory National Institute of Standards and Technology (NIST). Post-Quantum Cryptography | CSRC | CSRC, January 2017.

[16] David Pointcheval and Jacques Stern. Security Proofs for Signature Schemes. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Ueli Maurer, editors, *Advances in Cryptology — EUROCRYPT '96*, volume 1070, pages 387–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. Series Title: Lecture Notes in Computer Science.

[17] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[18] Dominique Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 755–784. Springer, 2015.