Design and Implementation of an Asynchronous Distributed Key Generation Protocol

Master Thesis

Markus Eggimann

Prof. Dr. Christian Cachin

Dr. Orestis Alpos & Mariarosaria Barbaraci

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

2023

Abstract

Distributed key generation (DKG) is a critical step during the initialization phase of distributed cryptosystems. The generation and distribution of cryptographic keys is a crucial prerequisite for many cryptographic primitives like digital signatures, threshold ciphers or pseudorandom number generators. In threshold cryptosystems, no single party is assumed to be trustworthy. However, one can assume that only a certain number of nodes, up to a threshold, is corrupted. Hence, collaboration between the participants is required to generate and distribute keys in a reliable and robust way, tolerating the presence of malicious parties. In this work, we present an asynchronous key generation protocol based on the well-known synchronous DKG protocol of Gennaro *et al.* [1]. Our main motivation is to build a protocol that assumes the availability of a total order broadcast primitive, and, hence, is suitable to run on existing blockchain infrastructure. We implement the protocol as part of the threshold cryptography service *Thetacrypt* [2].

Contents

1 Introduction

Nowadays, distributed computing systems are omnipresent. Examples include large-scale software services running in the cloud and blockchains running on thousands of computers all over the world. To guarantee confidentiality, integrity, authentication, and non-repudiation in such systems where one does not know who to trust is an active field of research in both industry and the academic world. Cryptography, and distributed cryptography in particular, provides tools and techniques to enforce security in distributed systems. Of particular relevance are threshold cryptosystems, where one assumes that a certain number of parties in the system is corrupted and controlled by an adversary.

Threshold cryptosystems provide cryptographic primitives like threshold signatures and threshold ciphers. Protocols implementing those primitives require a certain number of correct parties to collaborate to sign a message or decrypt it, respectively. Many of those cryptographic primitives assume the existence of a key or a key pair that is required to perform the cryptographic operation. Hence, the cryptosystems require a setup phase to generate and distribute those keys. The process that generates the keys is called *key generation*. There are two general approaches for key generation. The first approach assumes the existence of a trusted party that generates and distributes the keys. This approach cannot be used in environments where there is no trust in anyone. Hence, the second approach requires the parties to collaborate and follow a protocol to set up the keys. This is called *distributed key generation* (DKG). Protocols implementing DKG must be designed in a way that they can tolerate malicious parties.

Numerous protocols for DKG have been presented. In the literature, the solutions proposed by Pedersen [3] and later by Gennaro *et al.* [1] have gained the most interest and popularity. They became a starting point for subsequent research. However, both

protocols assume a synchronous system which makes them impractical for real-world use cases. Most real-world systems can be represented more realistically by an asynchronous model. That is the reason why, recently, several DKG protocols for asynchronous systems were published, like the one from Kokoris *et al.* [4] or from Abraham *et al.* [5].

However, neither of the mentioned protocols was designed for usage on an existing blockchain infrastructure. Hence, we designed a protocol that assumes the existence of a total order broadcast primitive, which is provided by modern blockchains. We took the synchronous protocol of Gennaro *et al.* as a starting point and modified it in a way that it works in an asynchronous system. We implemented the protocol and integrated it into a threshold cryptography service called *Thetacrypt*, which is developed by the Cryptology and Data Security research group at the University of Bern, Switzerland.

This thesis is structured as follows: Chapter 2 introduces core definitions and notions about distributed systems and cryptography used throughout this work, and presents the current state of research about DKG. Chapter 3 describes our asynchronous DKG protocol. Chapter 4 elaborates on the implementation and the testing of the protocol, while the final chapter 5 discusses the contribution of our work to the research field, as well as opportunities for future work.

2 Background

2.1 Definitions

2.1.1 Distributed Cryptosystems

A *distributed cryptosystem* consists of multiple, possibly independent, parties that collaboratively perform cryptographic operations. We denote as n the number of parties in such a cryptosystem. Many operations of a cryptosystem assume the existence of some key or, in the case of public key cryptography, the existence of a key pair. An example is *digital signatures* where the signer uses a secret key to sign a message. The recipient uses the corresponding public key to validate the signature on the message. Another example is encryption, where a public key is used to encrypt a message and the recipient of the message uses the corresponding secret key to decrypt the message.

Threshold cryptosystems are a subgroup of distributed cryptosystems where it is assumed that a certain number of parties, a threshold, are controlled by an adversary. The adversary tries to interfere with the operations of the cryptosystem. We let t denote the number of parties that are adversarially controlled. They might deviate from the protocol in any way and/or leak information to the adversary.

Threshold cryptosystems are based on *secret sharing*. That is, a secret – usually the secret key – is shared among the parties such that no single party knows the whole secret. Instead, each party knows just a part of it and the parties have to collaborate to recover and use the secret in cryptographic operations.

A well-known method for secret sharing is Shamir's scheme [6]. It is based on the

fact that a polynomial of degree k can be uniquely reconstructed when $k+1$ points on it are known. Hence, a dealer who wants to share a secret s chooses a polynomial $p(z) = \sum_{i=0}^{k} a_i z^i$ such that $a_0 = s$. The dealer then generates shares for each party p_j by computing $s_j = p(j)$ for $j = 1, \ldots, n$, and sends each share to the corresponding party. By construction, it requires the shares of at least $k+1$ parties to reconstruct the secret, using Lagrange interpolation. By setting $k = t$ we make sure that the adversary is unable to learn the secret because, for that, at least $t+1$ shares are required. If any $t+1$ parties should be able to reconstruct the secret, then $n > 2t$ is necessary.

Cryptographic protocols for threshold cryptosystems must work correctly and provide security even though the adversary controls a certain number of parties participating in the protocol. The protocols operating in such a setting usually define certain assumptions on the number of corrupted parties t with respect to the total number of parties n.

2.1.2 Distributed Key Generation

As already mentioned, threshold cryptosystems require cryptographic keys for many of their operations. Hence, key generation is an important problem in such systems. Distributed cryptosystems usually run through a setup phase during which the keys are generated and distributed.

There are two basic approaches to solving the key generation problem in distributed cryptosystems: the first approach assumes the existence of a special party that generates the key's parts and hands them out to the parties. The advantage is that this approach is very simple and fast: only a single party performs all the work and all other parties just wait to receive the key parts. The big disadvantage of this approach is that all parties have to trust this special party. That is why that party is often called the *trusted dealer*. Furthermore, the trusted dealer has to be online whenever keys have to be generated, making it a single point of failure.

The second approach is to let the parties collaboratively generate the key pair. The advantage of this approach is that there are no additional trust assumptions except for the number t of corrupted parties. The disadvantage of this approach is that it requires interaction, synchronization, and agreement between the parties, taking a considerable amount of time to complete. This approach is referred to as *distributed key generation* (DKG) and implementing such a protocol is the focus of this work.

2.1.3 Basic Models

All protocols that we will discuss in this work make certain assumptions about the environment they are designed for. In particular, the protocols use abstractions to describe time, communication, process crashes, and the capabilities of the adversary. In this section, we give a brief overview of these models. We do so by using the terminology and definitions from Cachin *et al.* [7].

Timing Model

The notion of time plays an important role in distributed cryptosystems and distributed systems in general. Depending on the model of time used in a distributed system, processes that are running in that system may or may not make certain time-related assumptions.

All systems fall in between two basic categories: *asynchronous* and *synchronous* systems. *Synchronous* systems allow processes to make assumptions about upper bounds for computation and networking delays. *Asynchronous* systems, on the other hand, have no notion of time. This means that processes are not allowed to make any timing-related assumptions, at all.

While the synchronous model is useful in theory, we usually encounter a hybrid model between synchronous and asynchronous in practice.

Communication Model

A distributed system is formed by multiple parties that collaborate. To allow the parties to work together, they need a way to communicate with each other. Hence, the parties are connected through one or multiple communication channels.

We distinguish between *point-to-point* channels and *broadcast* channels.

Point-to-point channels connect exactly two parties in the system. A party uses such a channel to send a message to a specific other party. Such channels might provide additional guarantees such as the following:

- *Reliable delivery*. No message is lost.
- *No duplication*. A message is delivered exactly once.
- *Authentication*. A message that is delivered by the recipient has been sent by the sender. No other party can intercept the channel and forge messages without getting noticed.

A *broadcast* channel connects every party with all other parties. Hence, a message that is posted on this channel will be delivered to all parties in the system. Broadcast channels might also have some additional properties:

- *Reliable delivery*. If a message is delivered by any process, it is eventually delivered by all processes.
- *Causal order*. If a broadcasted message m_1 caused another broadcasted message m_2, then all processes deliver the messages in order $m_1 \rightarrow m_2$.
- *Total order*. All messages are delivered by all processes in the very same (global) order.

Process Failure Model

Parties in distributed systems can fail in different ways:

- *Crash-stop failure*. A party stops executing steps forever. This is the simplest failure model.
- *Crash-recovery failure*. A party stops executing steps, but recovers later. It might have lost its internal state in the meantime, or its state might be outdated.
- *Arbitrary failure*, also known as byzantine failure. A party might deviate from a protocol in any way. This is the most general assumption.

The protocols we will see in this work assume arbitrary failures of parties.

Adversarial Model

Protocols in a distributed cryptosystem have to define the capabilities of the adversary. An adversary can be characterized by the following properties:

- *Computation power*. An adversary might be *computationally bounded* or *unbounded*. A computationally bounded adversary can be modeled by a probabilistic polynomial-time Turing machine.
- *Participation*. An adversary might be *passive* or *active*. A passive adversary shows eavesdropping behavior whereas an active adversary maliciously participates in the protocol, trying to disrupt it.
- *Behavior*. An adversary might be *static* or *adaptive*. A static adversary corrupts the parties at the beginning of the protocol, and the selection remains stable throughout the protocol run. An adaptive adversary chooses who to corrupt during the protocol run.
- *Communication*. An adversary might be *rushing*. This means that he speaks last in every round of communication after seeing all messages sent by the non-corrupted parties. This way he can try to bias the protocol output.

Most protocols that we will discuss assume a static, computationally bounded, active adversary that speaks last.

2.1.4 Cryptographic Assumptions

The distributed key generation protocols we are discussing in this work operate under the *discrete-log assumption*. That assumption states that it is computationally infeasible to compute discrete logarithms in certain algebraic groups. Of particular interest are groups modulo large primes.

The concept of a "large" prime can be formalized by introducing a security parameter k where the length of the binary representation of the prime number grows polynomially when increasing k. Nowadays, choosing $k \geq 2048$ is considered to be secure.

Let p be a large prime number for which there exists another large prime number q such that q divides $p - 1$. For any pair of prime numbers p and q, we define a subgroup G of order q in Z_p^*. We denote as g a generator of the group G. Given an element $y \in G$ we can write $y = g^x \mod p$ for $x \in [1, \ldots, q]$. x is then called the discrete logarithm of y with respect to g.

2.2 Related Work

2.2.1 Synchronous DKG Protocols

One of the first distributed key generation protocols for discrete log-based threshold cryptosystems was presented by Pedersen [3] in 1992. It is a *synchronous* protocol that provides resilience against a *static*, *computationally bounded*, *active* adversary that corrupts at most $t < n/2$ parties.

The protocol of Pedersen, as detailed by Gennaro *et al.* [1], works in rounds where in every round a party acts as a dealer to share a randomly chosen value that represents a part of the secret key. The other parties validate those shares against the commitments published by the dealing party and broadcast a complaint if the validation fails. The blamed party then reveals its shares such that the other parties can check them. The parties construct a set of qualified parties that behave correctly. The secret key is then composed of the shared values of the parties in that qualified set. The public key is reconstructed based on the values published by the parties in the qualified set.

The protocol uses *Feldman's verifiable secret sharing scheme* internally that provides computational security of the secret being shared. The protocol uses authenticated point-to-point channels to distribute the shares among the parties, and a broadcast channel to publish commitments, complaints and revelations.

Later, Gennaro *et al.* [1] found a flaw in the protocol of Pedersen that allows an attacker to influence the distribution of the generated keys such that it is not uniform. In the very same paper, they present a protocol, that is resistant to such an attack. We will refer to their protocol as *GJKR-DKG*. *GJKR-DKG* follows the same basic structure as the one from Pedersen, but it uses Pedersen's verifiable secret sharing scheme internally

that provides not computational but perfect (information-theoretic) secrecy of the shared secret. The protocol is *synchronous* and resistant to a *static*, *computationally bounded*, *active* adversary that corrupts up to $t < n/2$ parties. The protocol uses authenticated confidential point-to-point channels to distribute the shares among the parties, and a broadcast channel to publish commitments, complaints and revelations. We will describe *GJKR-DKG* in detail in the next section.

2.2.2 GJKR-DKG Protocol

GJKR-DKG works under the discrete-log assumption. In addition, Gennaro *et al.* assume the existence of a second group generator h of G where G is defined as explained in section 2.1.4. Note that the discrete log of h with respect to the primary group generator g must not be known to anyone to provide security.

The protocol operates in two phases. In the first phase, the *sharing* phase, each party shares a random value using Pedersen's verifiable secret sharing scheme. The party p_i chooses a random value z_i and two random polynomials $f_i(z) = a_{i0} + a_{i1}z + \cdots + a_{it}z^t$ and $f'_i(z) = b_{i0} + b_{i1}z + \cdots + b_{it}z^t$ of degree t such that $z_i = a_{i0} = f_i(0)$. p_i then broadcasts commitments $C_{ik} = g^{a_{ik}}h^{b_{ik}} \mod p$ for $k = 0, \ldots t$. After that, p_i computes shares $s_{ij} = f_i(j), s'_{ij} = f'_i(j)$ for each party p_j and sends them to each party using a direct encrypted point-to-point channel. Each party p_j verifies the shares it received from another party p_i by checking

$$g^{s_{ij}}h^{s'_{ij}} = \prod_{k=0}^{t}(C_{ik})^{j^k} \mod p \text{ for } i = 1, \ldots, n \tag{2.1}$$

If the check fails for some index i, the party that encountered the failing check broadcasts a complaint against party p_i. The blamed party answers the complaint by broadcasting the values (s_{ij}, s'_{ij}). The other parties can then verify whether party p_i answered the complaint with valid shares. Each party maintains a list of *disqualified* parties that either received more than t complaints or answered a complaint with values that do not pass the check. Each party constructs a set of non-disqualified parties *QUAL*. Gennaro *et al.* [1] show that, in the end, all honest parties build the same set. The shared values of the parties in *QUAL* will be used for constructing the shared secret key x. Note that x is never explicitly computed by any party, but it equals $x = \sum_{p_i \in \text{QUAL}} z_i$.

In the second phase, the *public key reconstruction* phase, the parties in *QUAL* reconstruct the public key $y = g^x \mod p$. Each party $p_i \in$ QUAL exposes $y_i = g^{z_i} \mod p$ using Feldman's verifiable secret sharing scheme by computing and broadcasting $A_{ik} = g^{a_{ik}} \mod p$ for $k = 1, \ldots, t$. The other parties verify those values by verifying that

$$g^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} \mod p \tag{2.2}$$

If the check fails for an index i, the verifying party p_j complains against p_i by broadcasting the values (s_{ij}, s'_{ij}). The other parties then reconstruct the share z_i of p_i by pooling the shares they got from p_i. Finally, the parties compute the public key $y = \prod_{i \in \text{QUAL}} y_i$ where $y_i = A_{i0} = g^{z_i}$.

2.2.3 Asynchronous DKG Protocols

The two DKG protocols we have seen in section 2.2.1 are both synchronous. Recent research presented DKG protocols that work under asynchronous assumptions.

Kokoris *et al.* [4] presented an asynchronous DKG in 2019. Their protocol builds upon asynchronous verifiable secret sharing that is used to build a common coin abstraction. That abstraction is then used to build a binary agreement abstraction that is, finally, used to build an asynchronous DKG protocol. Their protocol provides resilience against an *adaptive*, *computationally bounded*, *active* adversary that corrupts up to $t < n/3$ parties. They use authenticated confidential point-to-point channels and a reliable broadcast channel.

In 2021, Abraham *et al.* [5] presented another asynchronous DKG protocol that improved the communication complexity in comparison to the protocol of Kokoris *et al.* They managed to not require a binary agreement primitive, which is one reason why they achieved less communication complexity. Their protocol is resistant against a *static*, *computationally bounded*, *active* adversary that corrupts up to $t < n/3$ parties. They use authenticated confidential point-to-point channels and a reliable broadcast channel.

In 2022, Das *et al.* [8] came up with an asynchronous DKG protocol that used another approach to improve the communication cost compared to the protocol of Kokoris *et al.* Their protocol uses a new asynchronous complete secret sharing protocol internally to let each party share a part of the secret key. Their protocol then uses multiple reliable broadcasts to suggest sets of parties whose shares should be used for key generation. Finally, they run parallel asynchronous binary agreements to agree on the set of parties used for key generation. Their protocol is resistant against a *static*, *computationally bounded*, *active* adversary that corrupts up to $t < n/3$ parties. They implemented their protocol and ran some experiments to assess its performance by running a variable number of parties on a cloud service. Their protocol takes between 8.5 seconds (for $n = 16$) and 134 seconds (for $n = 64$) to generate keys.

3
Protocol Design

In this chapter, we present our asynchronous DKG protocol. It uses a total order broadcast channel and authenticated confidential point-to-point channels between all parties. It provides resilience against a *static*, *computationally bounded*, *active* adversary that corrupts up to $t < n/3$ parties.

3.1 Motivation

The motivation to design and implement an *asynchronous* DKG protocol is that asynchronous protocols are more relevant in practice than synchronous ones. We designed a *new* asynchronous protocol instead of implementing one of those presented in section 2.2.3 because we wanted to integrate a DKG protocol into an existing blockchain infrastructure. Blockchains provide a total order broadcast abstraction to order transactions, which in turn allows the implementation of a replicated state machine. Neither of the mentioned protocols was designed with the availability of a total order broadcast primitive in mind. Hence, we came up with our own protocol that exploits the properties the blockchain infrastructure, namely treating its total order broadcast primitive as a black box. Besides that, the asynchronous DKG protocols presented in section 2.2.3 lack the simplicity of *GJKR-DKG*. This work is an attempt to bring the simplicity of the original protocol to the asynchronous world.

3.2 The Protocol

The main idea behind our protocol is to take the synchronous *GJKR-DKG* as a base and adapt it to the asynchronous world. Hence, the high-level steps of our protocol are the same as in the original protocol. Our protocol, too, is divided into a *sharing phase* and a *public key reconstruction phase*. Be aware that in the following sections, the sharing and the public key reconstruction phases are described separately from each other. But, when it comes to implementing them, they cannot be that strictly separated. Some logic of the reconstruction phase depends on the state that is built during the sharing phase, and some events that are described in the public key reconstruction phase might already occur while a party is still doing operations that are described in the sharing phase. As a consequence, the logic of both phases interleave in the actual implementation.

3.2.1 Sharing Phase

Recall the sharing phase of *GJKR-DKG*:

S.1 Each party p_i acts as a dealer to choose and share parts of the secret key z_i.

S.2 The other parties validate the shares and complain against the dealing party in case it sent invalid shares.

S.3 The blamed party defends itself by publishing the shares it sent to the blaming party.

S.4 The other parties validate the shares published by the blamed parties.

S.5 The parties build up a set *QUAL* of qualified parties whose key shares should be used for key generation of the secret key x and reconstruction of the public key y based on x.

S.6 Each party computes its share of the secret key x_i based on the shares they received from the members of *QUAL*.

Our protocol follows the same steps. However, because we operate in an asynchronous setting, the steps cannot be done in the same way. We use three key ideas to deal with the challenges imposed by the asynchrony.

Consider step S.2 above: during the sharing phase of the original protocol, the parties complain in case they receive an invalid share. They implicitly assume an upper time bound on how long it takes a party to get the shares, validate them, and raise a complaint, if necessary. In the asynchronous world, such an assumption cannot be made.

Hence, we need another way to tell whether or not a party complained against another party which leads us to the *first key idea* of our protocol: instead of letting the parties

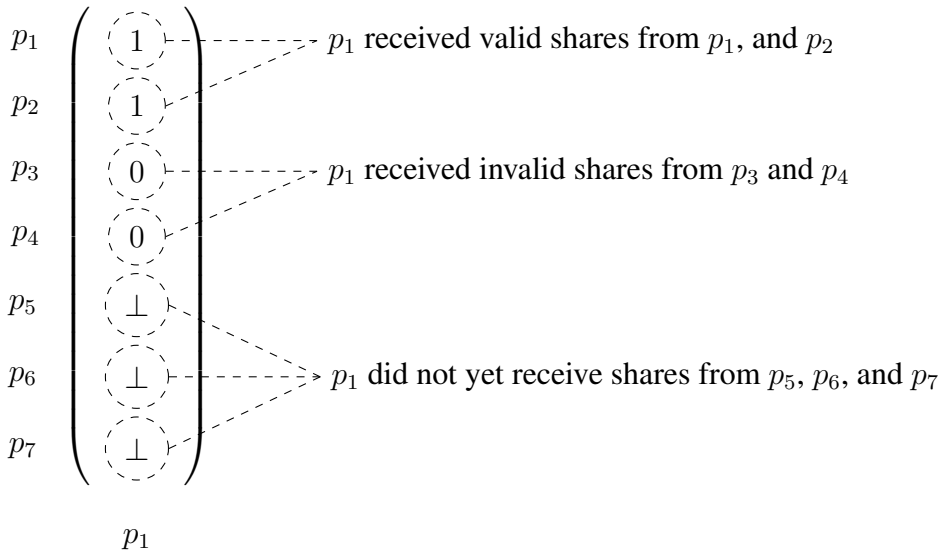

Figure 3.1. An example verification vector of party p_1 in a system with $n = 7$ parties.

complain immediately, we let them build up a local vector of size n holding their votes on the shares they received from the other parties. We call such a vector a *verification vector*. Initially, all entries in the verification vector are empty, denoted by $\perp$. This means, that the party did not receive a share from any party yet. If a party p_i receives a *valid* share from party p_j it sets the j-th entry in its verification vector to 1. If it received an invalid share from p_j, instead, it would set the j-th entry to 0. Figure 3.1 shows an example of a verification vector with some entries already filled in.

As soon as a party has at least $(n - t)$ non-empty entries in its local verification vector, it broadcasts its current verification vector and all subsequent updates to it. This exploits the *second key idea* of our design: we use a total order broadcast primitive to allow the parties to share a common view on who sent valid or invalid shares to whom, without revealing the shares themselves.

Recall that the goal of the sharing phase is to decide which parties will be part of the qualified set *QUAL*. In *GJKR-DKG* they do so by making an implicit assumption on the time it takes for the honest parties to (a) send, receive, and validate all shares, (b) send, receive and answer complaints, and (c) validate the answers on the complaints. Again, this is not possible in the asynchronous setting.

Hence, we use a different strategy that represents the *third key idea* of our design: the parties use the verification vectors they receive through the broadcast to build a so-called *verification matrix* that has $n \times n$ entries. Each column vector of the matrix corresponds to a verification vector of a single party. Hence, the i-th column in the verification matrix of party p_i is equivalent to the verification vector it builds when it receives shares from other parties. The j-th column in the verification matrix of party p_i is the verification

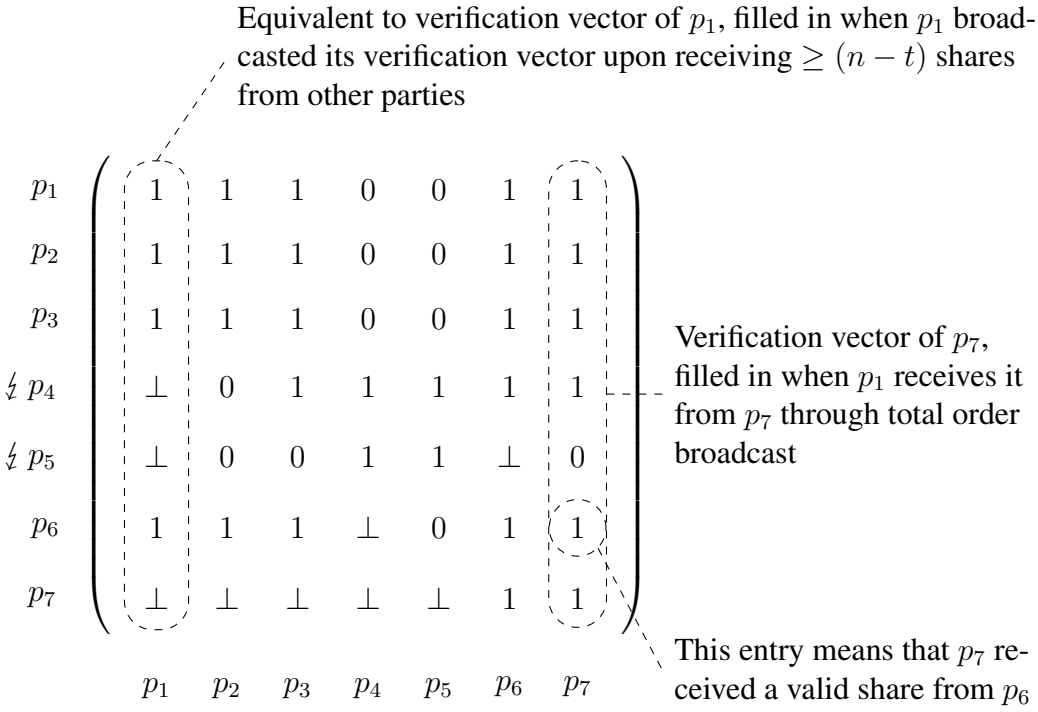

Figure 3.2. The verification matrix of party p_1 in a system with $n = 7$ parties. Parties p_4 and p_5 are corrupted.

vector it received from party p_j through the total order broadcast. This includes the verification vector p_i publishes itself. Figure 3.2 shows an example of a verification matrix and illustrates the meaning of the different parts of the matrix.

If a party p_i receives a verification vector V from another party p_j with the entry at index i set to 0, meaning that p_j complains about p_i, it broadcasts the shares it sent to p_j before to all parties. The other parties can then verify whether p_i sent correct shares to p_j and set the corresponding entry in their verification matrix.

We want to emphasize that all entries in the verification matrix of a party p_i are only modified upon reception of messages through the total order broadcast channel, and these messages can be either updated verification vectors or shares in response to complaints. The way the verification matrix is computed ensures that all correct processes end up with the same entries in their local verification matrix.

The second last step S.5 of the sharing phase consists of deciding which parties will be part of *QUAL*. For this, after each change to the verification matrix we try to find a set of $2t + 1$ candidate parties out of the total n parties such that the corresponding sub-matrix of the verification matrix has only '1' entries. This means that all the chosen parties received the shares from all other chosen parties – or at least claim they have, in case they are malicious. Constructing *QUAL* like this ensures that every party $p_i \in$ *QUAL*

$$
\begin{array}{r}
p_1 \\ p_2 \\ p_3 \\ \lightning\ p_4 \\ \lightning\ p_5 \\ p_6 \\ p_7
\end{array}
\begin{pmatrix}
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 1 \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 1 \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 1 \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & \mathbf{1} & 1 \\
\bot & 0 & 0 & 1 & 1 & \bot & 0 \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & \mathbf{1} & 1 \\
\bot & \bot & \bot & 1 & \bot & 1 & 1
\end{pmatrix}
\begin{array}{c}
\\ \\ \\ \\ \\ \\ \\
\end{array}
$$

$$
\begin{array}{ccccccc}
p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7
\end{array}
$$

Figure 3.3. An example for the construction of the qualified set out of the verification matrix of p_1. Parties p_4 and p_5 are corrupted.

will be able to construct its part of the secret key x_i in the last step S.6 of the sharing phase. x_i is defined as $x_i = \sum_{p_j \in QUAL} s_{ji} \mod p$. We will eventually be able to find a set of parties that matches the condition since the $2t + 1$ honest parties eventually receive the shares from each other. As soon as we find a candidate set $\mathcal{S}$ that matches the condition, we set $QUAL \leftarrow \mathcal{S}$.

The algorithm that produces the candidate sets must generate them in some deterministic order that is the same for all (honest) parties. If this holds, and because all correct parties computed the same verification matrix in steps S.2 - S.4, all correct parties end up with the same set *QUAL*.

Figure 3.3 illustrates the construction of the qualified set. Parties p_1 to p_4 and p_6 are a candidate set because they all received the shares from each other. Note that, in this example, the corrupted party p_4 will be part of *QUAL*, whereas the honest party p_7 will not be included. This does not matter as the other parties in *QUAL* will be able to compensate for future possibly faulty behavior of p_4. We will briefly discuss the resilience of the protocol later in section 3.3.

The pseudocode of the sharing phase is listed in algorithms 1 and 2. The pseudocode is written in an event-based reactive programming style, where all logic is executed in response to events that happen. These events might either be messages that are delivered or a predicate on the local state of a party that evaluates to `true`. Furthermore, the algorithms use helper functions that are listed in section 3.2.3.

Algorithm 1 Asynchronous distributed key generation (sharing phase, process $p_i \in \mathcal{P}$)

state
  commitments $\leftarrow [\perp]^n$: associative set of commitments
  shares $\leftarrow [\perp]^n$: associative set of shares
  verificationMatrix $\leftarrow [\perp]^{n \times n}$: the verification matrix
  verificationVector $\leftarrow [\perp]^n$: the verification vector
  revelations $\leftarrow [\perp]^{n \times n}$: associative set holding revelations of shares
  QUAL $\leftarrow \emptyset$: set of qualified parties
  verificationSent $\leftarrow 0$: before verification vector sent

upon event *init*()
  $z_i \xleftarrow{\$} \mathbb{Z}_q$
  choose random polynomial $f_i(z)$, $f'_i(z)$ over $\mathbb{Z}_q$ of degree t such that:
    $f_i(z) \leftarrow a_{i0} + a_{i1}z + \cdots + a_{it}z^t$
    $f'_i(z) \leftarrow b_{i0} + b_{i1}z + \cdots + b_{it}z^t$
    $z_i = a_{i0} = f_i(0)$
  compute $C_{ik} \leftarrow g^{a_{ik}} h^{b_{ik}} \mod p$ for $k = 0, \ldots, t$
  atomically broadcast message [COMMIT, $\{C_{i0}, \ldots, C_{it}\}$]
  for $j = 1, \ldots, n$ **do**
    compute $s_{ij} \leftarrow f_i(j) \mod q$
    compute $s'_{ij} \leftarrow f'_i(j) \mod q$
    send [SHARE, (s_{ij}, s'_{ij})] to $p_j \in \mathcal{P}$

upon atomically delivering a message [COMMIT, $\{C_{j0}, \ldots, C_{jt}\}$] from p_j
  such that *commitments* $[j] = \perp$ **do**
  commitments $[j] \leftarrow \{C_{j0}, \ldots, C_{jt}\}$
  if *shares* $[j] \neq \perp$ **then**
    verificationVector $[j] \leftarrow$ *isValidShare*(*shares* $[j]$, *commitments* $[j]$)
    if *verificationSent* $= 1$ **then**
      atomically broadcast [VERIFICATION, *verificationVector*]

Algorithm 2 Asynchronous distributed key generation (sharing phase, process $p_i \in \mathcal{P}$) (cont.)

27: **upon** delivering a message $[\text{SHARE}, (s_{ji}, s'_{ji})]$ from p_j such that *shares*$[j] = \perp$ **do**
28: *shares* $[j] \leftarrow (s_{ji}, s'_{ji})$
29: **if** *commitments* $[j] \neq \perp$ **then**
30: *verificationVector* $[j] \leftarrow$ *isValidShare*(*shares* $[j]$, *commitments* $[j]$)
31: **if** *verificationSent* $= 1$ **then**
32: atomically broadcast $[\text{VERIFICATION}, \textit{verificationVector}]$

33: **upon** *verificationSent* $= 0 \wedge \left|\{p_j \in \mathcal{P} | \textit{verificationVector}\,[j] \neq \perp\}\right| \geq n - t$ **do**
34: atomically broadcast $[\text{VERIFICATION}, \textit{verificationVector}]$
35: *verificationSent* $\leftarrow 1$

36: **upon** atomically delivering a message $[\text{VERIFICATION}, V_j]$ from p_j
such that *isValidVerificationVector*$(V_j) = 1$ **do**
37: **for** $k = 1, \ldots, n$ **do**
38: **if** *verificationMatrix* $[k]\,[j] \neq 1$ **then**
39: *verificationMatrix* $[k]\,[j] \leftarrow V_j\,[k]$
40: **if** $V_j\,[i] = 0 \wedge$ *commitments* $[i] \neq \perp$
$\wedge\, p_i$ did not already reveal (s_{ij}, s'_{ij}) (see line 41) **then**
41: atomically broadcast $[\text{REVEAL}, (s_{ij}, s'_{ij})]$

42: **upon** atomically delivering a message $[\text{REVEAL}, (s_{jl}, s'_{jl})]$ from p_j **do**
43: *revelations* $[j]\,[l] \leftarrow (s_{jl}, s'_{jl})$
44: **let** *isValidRevelation* $\leftarrow$ *isValidShare*(*revelations* $[j]\,[l]$, *commitments* $[l]$)
45: *verificationMatrix* $[j]\,[l] \leftarrow$ *isValidRevelation*
46: **if** $l = i$ **then**
47: *verificationVector* $[j] \leftarrow$ *isValidRevelation*
48: **if** *verificationSent* $= 1$ **then**
49: atomically broadcast $[\text{VERIFICATION}, \textit{verificationVector}]$

50: **upon** $\exists\, \mathcal{S}$ where $\mathcal{S} \leftarrow \{p_j \in \mathcal{P} \mid p_j$ has shares from all $p_k \in \mathcal{S}\} \wedge |\mathcal{S}| = 2t + 1$ **do**
51: *QUAL* $\leftarrow \mathcal{S}$
52: **if** $p_i \in$ *QUAL* **then**
53: **let** $X_i \leftarrow \emptyset$
54: **for** $p_v \in$ *QUAL* **do**
55: **let** $(s_{vi}, s'_{vi}) \leftarrow$ *shares*$[v]$
56: $X_i \leftarrow X_i \cup \{s_{vi}\}$
57: compute secret key share x_i as $x_i \leftarrow \sum_{s_{li} \in X_i} s_{li}$
58: start reconstruction phase of the protocol (see algorithm 3)

3.2.2 Public Key Reconstruction Phase

Recall the *public key reconstruction phase* of *GJKR-DKG*:

R.1 Each party p_i in *QUAL* acts as a dealer to share the value g^{z_i}, where z_i is the secret from the sharing phase. We will refer to these shares as *public key shares*.

R.2 The other parties validate the public key shares and complain against the dealing party in case it sent invalid public key shares.

R.3 The blaming party reveals the secret key shares it got from the party it complained about.

R.4 The other parties validate those shares. In case the complaint is valid the other parties reconstruct the secret for the blamed party.

R.5 Finally, the parties use the public key shares to recover the public key y.

Again, our protocol does basically the same. As a consequence, we face similar challenges imposed by asynchrony as in the first phase. The main problem in this phase is how we can decide whether or not we should reconstruct the secret of a party. To solve this problem, we follow a similar approach as in the first phase: we let the parties validate the public key shares they received from the others through total order broadcast, build a verification vector, and broadcast those vectors. The parties use those vectors to build a verification matrix to decide which parties (in *QUAL*) behaved correctly in phase 1 (their shares pass equation 2.1) *and* phase 2 (their public key shares pass equation 2.2). However, the naïve approach of just letting the parties in *QUAL* vote falls short. We need $2t + 1$ votes for a party to get a majority of correct parties voting in favor of it. But we only have a total of $2t + 1$ parties in *QUAL*, and up to t of them might be malicious. Hence, we also let the parties outside of *QUAL* participate in the vote.

There is a subtle difference between the voting in the sharing phase and the one in the public key reconstruction phase: in the former, the blamed party reveals its shares. In the latter, it is the blaming party that reveals the shares it got from the party it complains about. As a consequence, the structure of the verification vectors is a bit different. Instead of just broadcasting a 0 entry to indicate a complaint, the complaining party broadcasts the shares it considers invalid within the verification vector. The other parties are then able to validate the shares themselves. Figure 3.4 shows an example of a verification vector with embedded shares in case of a complaint.

If a party confirms (locally) that a complaint is justified, it knows that the shares for the blamed party have to be reconstructed. It broadcasts the shares it got from the blamed party and waits for the other (correct) parties to do the same. As soon as it gets more than t shares, it is able to reconstruct the secret of the blamed party locally by doing Lagrange interpolation using the shares it collected.

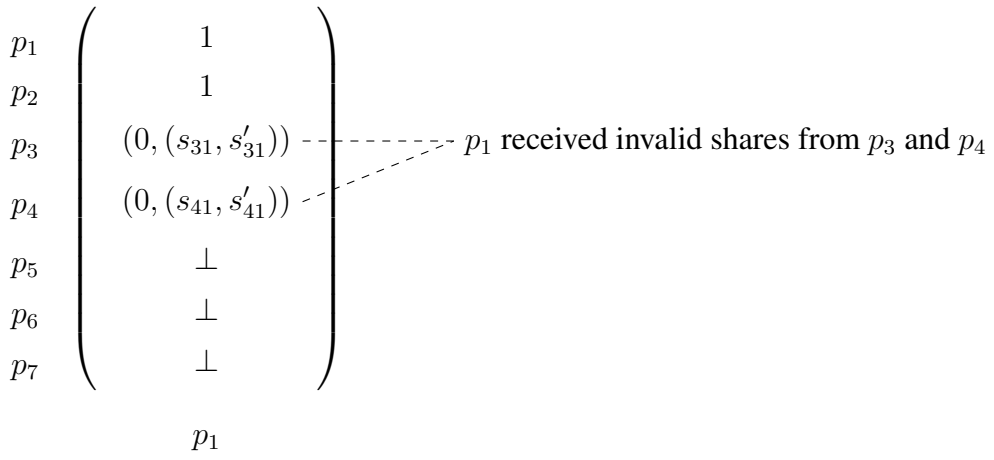

Figure 3.4. An example verification vector of party p_1 in a system with $n = 7$ parties. p_1 complains about the shares it received from parties p_3 and p_4 by embedding the shares it got from p_3, and p_4 respectively.

The process of validating the public key shares embedded into a verification vector involves some subtleties. A party p_i needs the commitments $C_{j0}, \ldots, C_{jt}$ as well as the public key shares $A_{j0}, \ldots, A_{jt}$ from the blamed party p_j to perform the validation of a complaint. However, it might be the case that one or both preconditions are not fulfilled:

- In case the verifying party does not have the public key shares $A_{j0}, \ldots, A_{jt}$ from the blamed party, the original complaint – delivered through total order broadcast – cannot be justified. The complaining party must have received the public key shares from the blamed party through total order broadcast beforehand. Otherwise, it cannot raise a valid complaint. Hence, all other parties received the public key shares, too. Hence, it is safe to just ignore the complaint.
- In case the verifying party does not have the commitments $C_{j0}, \ldots, C_{jt}$ from a blamed party, the blamed party cannot be a member of *QUAL*. Hence, it is safe to ignore the complaint because the inputs of the blamed party are not used for public key reconstruction anyway.

The broadcasting of reconstruction shares, too, contains a small detail that requires further explanation. Assume that a party p_j wants to reveal a share of a party p_k that got a justified complaint. It can be the case that p_j did not receive the shares (s_{kj}, s'_{kj}) from p_k, e.g. because the connection between both parties is slow. It is safe for p_j to do nothing in that case because, by construction of *QUAL*, there are enough other correct parties that will be able to reveal the shares they got from p_k to allow secret reconstruction.

When a party p_j receives reconstruction shares (s_{kl}, s'_{kl}) from p_l to reconstruct the secret of p_k, it has to validate them against the commitments $C_{k0}, \ldots, C_{kt}$ it got from p_k

through total order broadcast. If p_j has not yet received the commitments it indicates misbehavior of p_l. If p_l was correct it would have broadcasted the reconstruction shares upon receiving a valid complaint against p_k through total order broadcast. A (valid) complaint can only be raised by a correct party if it received the commitments of p_k before through total order broadcast. This means that p_j would also have them. Hence, it is safe for p_j to simply ignore the reconstruction shares it got from p_l.

As said earlier, the parties use the verification vectors they received through the broadcast channel to build a verification matrix. Like in the first phase, the correct parties will end up with an identical verification matrix, because they fill the entries upon receiving information through the total order broadcast channel. Evaluating the rows of the verification matrix, counting the 1s per row, we can decide whether the secret of a party does *not* have to be reconstructed. This is the case when a party got at least $2t + 1$ votes. We call those parties validated parties. As soon as we have at least $t + 1$ validated parties we can trigger secret reconstruction for the remaining t parties in *QUAL*. This allows us to deal with an adversary that does not send any messages throughout the second phase.

As soon as we get all public key shares of all parties in *QUAL* – either by directly using the values received from the party or through reconstruction – we can compute the public key and complete the protocol run. Each (honest) party in *QUAL* now holds a share of the secret key and the public key.

The detailed steps of the public key reconstruction phase are described in the algorithms 3 and 4. Again, the algorithms use helper functions that are listed in section 3.2.3.

Algorithm 3 Asynchronous distributed key generation (pk reconstruction phase, process p_i)

59: **state**
60: *QUAL* $\leftarrow$ *QUAL* from sharing phase: set of qualified parties
61: $n_{QUAL} \leftarrow |QUAL|$: size of the qualified set
62: *pubKeyShares* $\leftarrow [\perp]^{n_{QUAL}}$: associative set of public key shares
63: *pubKeyVerificationMatrix* $\leftarrow [\perp]^{n_{QUAL} \times n}$: verification matrix
64: *needsReconstruction* $\leftarrow [\perp]^{n_{QUAL}}$: bit-vector indicating whether public key share for a party has to be reconstructed
65: *reconstructionShares* $\leftarrow [\emptyset]^{n_{QUAL}}$: multiset of reconstruction shares
66: *reconstructed* $\leftarrow [\perp]^{n_{QUAL}}$: associative set of reconstructed public key shares
67: *pubKeyVerificationSent* $\leftarrow 0$ // *before verification vector sent*
68: *reconstructedParties* $\leftarrow \emptyset$: set of parties the public key share was reconstructed for
69: *validatedParties* $\leftarrow \emptyset$: set of parties that sent correct public key share

70: **upon event** *init*()
71: **if** $p_i \in QUAL$ **then**
72: compute $A_{ik} \leftarrow g^{a_{ik}} \mod p$ for $k = 0, \ldots, t$
73: atomically broadcast [PUBKEYSHARES, $\{A_{i0}, \ldots, A_{it}\}$]

74: **upon** atomically delivering message [PUBKEYSHARES, $\{A_{j0}, \ldots, A_{jt}\}$] from $p_j \in QUAL$ **do**
75: *pubKeyShares*$[j] \leftarrow \{A_{j0}, \ldots, A_{jt}\}$
76: *pubKeyVerificationMatrix*$[j][i] \leftarrow$ isValidPubKeyShare(*shares*$[j]$, *pubKeyShares*$[j]$)
77: **if** *pubKeyVerificationSent* $= 1$ **then**
78: **let** $V_i \leftarrow$ *buildPubKeyVerificationVector*()
79: atomically broadcast [PUBKEYVERIFICATION, V_i]

80: **upon** *pubKeyVerificationSent* $= 0$
$\wedge \left|\{p_j \,|\, p_j \in QUAL, pubKeyVerificationMatrix[j][i] \neq \perp\}\right| \geq n - t$ **do**
81: **let** $V_i \leftarrow$ *buildPubKeyVerificationVector*()
82: atomically broadcast [PUBKEYVERIFICATION, V_i]
83: *pubKeyVerificationSent* $\leftarrow 1$

84: **upon** atomically delivering message [PUBKEYVERIFICATION, V_j] from p_j **do**
such that *isValidPubKeyVerificationVector*$(V_j) = 1$ **do**
85: **for** $k = 1, \ldots, n_{QUAL}$ **do**
86: **if** *pubKeyVerificationMatrix*$[k][j] \neq 1$ **do**
87: *pubKeyVerificationMatrix*$[k][j] \leftarrow V_j[k]$
88: **if** $V_j[k] = \{0, (s_{kj}, s'_{kj})\} \wedge$
pubKeyShares$[k] \neq \perp \wedge$ *commitments*$[k] \neq \perp \wedge$
isValidComplaint$((s_{kj}, s'_{kj}), pubKeyShares[k], commitments[k]) = 1$ **then**
89: *needsReconstruction*$[k] \leftarrow 1$
90: **if** *shares*$[k] \neq \perp$ **do**
91: atomically broadcast [RECONSTRUCTIONSHARE, *shares*$[k]$]

Algorithm 4 Asynchronous distributed key generation (pk reconstruction phase, process p_i) (cont.)

92: **upon** atomically delivering message [RECONSTRUCTIONSHARE, (s_{kj}, s'_{kj})] from p_j such that *commitments*$[k] \neq \perp$ **do**
93: **if** *isValidShare*$((s_{kj}, s'_{kj}, \textit{commitments}[k])) = 1$ **then**
94: *reconstructionShares*$[k] \leftarrow$ *reconstructionShares*$[k] \cup (s_{kj}, s'_{kj})$

95: **upon** $|\textit{reconstructionShares}[j]| > t \wedge \textit{needsReconstruction}[j] = 1$ for any $p_j \in$ *QUAL* **do**
96: *reconstructed*$[j] \leftarrow$ reconstruct(*reconstructionShares[j]*)
97: *reconstructedParties* $\leftarrow$ *reconstructedParties* $\cup\, p_j$

98: **upon** *cannotBePubKeyDisqualified*$(p_j) = 1$ for any $p_j \in$ *QUAL* **do**
99: *validatedParties* $\leftarrow$ *validatedParties* $\cup \{p_j\}$

100: **upon** $|\textit{validatedParties}| = t + 1$ **do**
101: **for** $p_j \in \textit{QUAL} \backslash (\textit{reconstructedParties} \cup \textit{validatedParties})$ **do**
102: *needsReconstruction*$[j] \leftarrow 1$
103: **if** *shares*$[j] \neq \perp$ **do**
104: atomically broadcast [RECONSTRUCTIONSHARE, *shares*$[j]$]

105: **upon** *reconstructedParties* $\cup$ *validatedParties* $=$ *QUAL* **do**
106: **let** $Y \leftarrow \emptyset$
107: **for** $p_r \in$ *reconstructedParties* **do**
108: **let** $z_r \leftarrow$ *reconstructed*$[r]$
109: $Y \leftarrow Y \cup \{g^{z_r}\}$
110: **for** $p_v \in$ *validatedParties* **do**
111: **let** $\{A_{v0}, \ldots, A_{vt}\} \leftarrow$ *pubKeyShares*$[v]$
112: $Y \leftarrow Y \cup \{A_{v0}\}$
113: compute public key y as $y \leftarrow \prod_{y_j \in Y} y_j \mod p$
114: terminate

3.2.3 Helper Functions

The algorithms in this section contain some helper functions that are used by the protocol.

Algorithm 5 Asynchronous distributed key generation (helper functions, process $p_i \in \mathcal{P}$)

115: **function** *isValidShare*$((s_{lj}, s'_{lj}), [C_{l0}, \ldots, C_{lt}]) \rightarrow \{0, 1\}$

116: **return** $\begin{cases} 0, & \text{if } g^{s_{lj}} h^{s'_{lj}} \neq \prod_{k=0}^{t} (C_{lk})^{j^k} \mod p \\ 1, & \text{if } g^{s_{lj}} h^{s'_{lj}} = \prod_{k=0}^{t} (C_{lk})^{j^k} \mod p \end{cases}$

117: **function** *isValidPubKeyShare*$((s_{lj}, s'_{lj}), [A_{l0}, \ldots, A_{lt}]) \rightarrow \{0, 1\}$

118: **return** $\begin{cases} 0, & \text{if } g^{s_{lj}} \neq \prod_{k=0}^{t} (A_{lk})^{j^k} \mod p \\ 1, & \text{if } g^{s_{lj}} = \prod_{k=0}^{t} (A_{lk})^{j^k} \mod p \end{cases}$

119: **function** *isValidComplaint*$((s_{lj}, s'_{lj}), [A_{l0}, \ldots, A_{lt}], [C_{l0}, \ldots, C_{lt}]) \rightarrow \{0, 1\}$

120: **return** *isValidShare*$((s_{lj}, s'_{lj}), [C_{l0}, \ldots, C_{lt}]) = 1$
$\wedge$ *isValidPubKeyShare*$((s_{lj}, s'_{lj}), [A_{l0}, \ldots, A_{lt}]) = 0$

121: *// A party cannot be disqualified if at least* $2t + 1$ *parties voted for it*

122: **function** *cannotBePubKeyDisqualified*$(p_j \in \mathcal{P}) \rightarrow \{0, 1\}$

123: **let** *confirmCount* $\leftarrow$ *countPubKeyVerificationOccurrences*$(1, j)$

124: **if** *confirmCount* $> 2t$ **then**

125: **return** 1

126: **return** 0

127: **function** *countPubKeyVerificationOccurrences*$(val \in \{0, 1\}, j \in \{1, \ldots, n\}) \rightarrow c \in \mathbb{N}$

128: **let** *count* $\leftarrow 0$

129: **for** $l = 1, \cdots, n$ **do**

130: **if** *pubKeyVerificationMatrix* $[j]\,[l] =$ *val* **then**

131: *count* $\leftarrow$ *count* $+ 1$

132: **return** *count*

133: **function** *isValidVerificationVector*$(V_j \in [\{\perp, 0, 1\}]^n) \rightarrow \{0, 1\}$

134: **return** $\begin{cases} 1, & \text{if } |\{v \in V_j | v \neq \perp\}| \geq n - t \\ 0, & \text{otherwise} \end{cases}$

Algorithm 6 Asynchronous distributed key generation (helper functions, process $p_i \in \mathcal{P}$) (cont.)

135: **function** *isValidPubKeyVerificationVector*($V_j \in \left[\{\perp, \{0, (s_{ji}, s'_{ji})\}, 1\}\right]^{n_{QUAL}}$) $\rightarrow \{0, 1\}$

136: **return** $\begin{cases} 1, & \text{if } |\{v \in V_j | v \neq \perp\}| \geq t + 1 \\ 0, & \text{otherwise} \end{cases}$

137: **function** *buildPubKeyVerificationVector*() $\rightarrow \{1, \{0, (s_{ji}, s'_{ji})\}, \perp\}^{n_{QUAL}}$

138: **let** $V_i \leftarrow [\perp]^{n_{QUAL}}$

139: **for** $j = 1, \ldots, n_{QUAL}$ **do**

140: **if** *pubKeyVerificationMatrix*$[j][i] = 1$ **then**

141: $V_i[j] \leftarrow 1$

142: **else if** *pubKeyVerificationMatrix*$[j][i] = 0$ **then**

143: $V_i[j] \leftarrow \{0, \textit{shares}[j]\}$

144: **else**

145: $V_i[j] \leftarrow \perp$

146: **return** V_i

147: **function** *reconstruct*($\{(s_{jk}, s'_{jk})\}$) $\rightarrow z_j \in \mathbb{Z}_q$

148: **let** $z_j \leftarrow$ perform Lagrange interpolation using $\{(s_{jk}, s'_{jk})\}$

149: **return** z_j

3.3 Security

We will not give formal proof of the security of our protocol. Instead, we will reason informally why our protocol maintains the same level of security and leave formal security proof as a subject for future work. Gennaro *et al.* define the following security property a secure DKG protocol has to fulfill:

> No information on [the secret] x can be learned by the adversary except for what is implied by the value $y = g^x \mod p$.

Furthermore, they define a requirement for the distribution of the generated secret key x:

> x is uniformly distributed in $\mathbb{Z}_q$, and, hence, y is uniformly distributed in the subgroup generated by [the generator] g

These requirements must be fulfilled by our protocol, as well. We assume that the adversary, in the worst case, controls up to t parties, where $t < n/3$. W.l.o.g. let $p_1, \ldots, p_t$ be the corrupted parties.

3.3.1 Sharing Phase

Recall the sharing phase of our protocol. During this phase, each party commits to a secret and shares it using Shamir's secret sharing. The shares are validated by the other parties. They hold a vote to agree on a set of qualified parties *QUAL* whose inputs should be used for key generation.

During the sharing phase, the adversary could try to *learn the secrets being shared by the correct parties*. The adversary can force other parties to reveal their shares by letting the corrupted parties complain about honest ones. Let p_c be a party that is correct and blamed by the adversary. p_c will react to the complaints by revealing the shares (s_{ck}, s'_{ck}) where $k = 1, \ldots, t$. The adversary will obtain at most t shares of the secret being shared by p_c. Due to this fact, and because honest parties will never complain against other honest parties, the adversary is not able to obtain enough shares to reconstruct the secret of p_c.

The adversary could try to *influence the distribution of the secret key* by trying to disqualify a correct party. However, for a party to be disqualified at least $t+1$ parties must vote against it. That is, at least one correct party must vote against another correct party, which will never happen. Hence, the adversary cannot force a party to be disqualified. However, if the adversary behaves correctly during the sharing phase some, or all, of the corrupted parties could become members of *QUAL*, effectively preventing some honest parties from being included in *QUAL*. Nevertheless, because we have $2t + 1$ parties in *QUAL* we have at least $t + 1$ honest parties in *QUAL*, whereas the contribution of a single honest party will suffice to guarantee the generation of an unbiased key pair.

At the end of the sharing phase, all honest parties end up with the same set *QUAL*. They build *QUAL* based on the verification matrix, which they computed locally based on the verification vectors published through total order broadcast by the other parties. Hence, all honest parties compute the same verification matrix, and, as a consequence, the same set *QUAL*. All members within *QUAL* got votes from at least $2t + 1$ parties, which means they got at least $t + 1$ votes from correct parties. There are $2t + 1$ parties in *QUAL* of which at least $t + 1$ are correct.

3.3.2 Public Key Reconstruction Phase

Recall the public key reconstruction phase of our protocol. The parties in *QUAL* broadcast their shares of the public key. The other parties validate those shares and hold a vote on whether the secret of a party has to be reconstructed using the information shared during the first phase, or not.

During this phase, the adversary could try to *force the reconstruction of the secret of a correct party*. Because all parties would expose the shares they got from that party during reconstruction, the adversary would learn the secret from that party. Let p_c again denote a correct party and p_a a corrupted party. The adversary has two ways to force the

other parties to reconstruct the secret of p_c:

- He could reveal some shares $(\overline{s}_{ca}, \overline{s}'_{ca})$ and claim he got them from p_c. Those shares $(\overline{s}_{ca}, \overline{s}'_{ca})$ must satisfy equation 2.1 but not equation 2.2 and must be different from the real (s_{ca}, s'_{ca}) that p_a received from p_c. Coming up with such shares is infeasible under the discrete log assumption, and, hence, the adversary cannot force correct parties to reveal the shares they got from p_c in that way.

- The adversary could publish valid public key commitments of all parties he controls before any other (honest) party does so. That way, t (malicious) parties publish correct public key shares. If one honest party then reveals a valid public key share, too, the other parties would then start reconstructing the secret from the remaining t (honest) parties in *QUAL*. As a consequence, the adversary can learn the secrets of t honest parties. But there is one honest party left in *QUAL* whose secret will not be reconstructed. The contribution of that single honest party is sufficient to guarantee the generation of an unbiased secret key.

The adversary could try to *attack termination of the protocol*. He could do so by behaving correctly during the sharing phase but showing malicious behavior during the second phase. For instance, he could remain silent and not send any public key shares, verification vectors and reconstruction shares. However, since there are at least $t + 1$ correct parties in *QUAL* they are eventually able to reconstruct the secrets of the malicious parties in *QUAL*.

In both phases, the adversary could also try *to forge messages*, hereby *impersonating other parties*. However, the authenticated point-to-point channels used in our protocol guard against message alteration and identity theft. Furthermore, the total order broadcast ensures that all correct parties deliver the messages through that channel in the same order. As a consequence, the adversary cannot trick an honest party into having another view of the state of the protocol than the other (correct) parties.

4 Protocol Implementation

4.1 Thetacrypt

Thetacrypt [2] is a software service, written in Rust, that implements several threshold cryptography primitives like ciphers, signatures, and coins. It aims to provide threshold cryptography as a service. The service is developed by the Cryptology and Data Security research group [9] at the University of Bern, Switzerland.

The service is split into three layers, depicted in figure 4.1. The *service layer* provides an RPC API to host applications. Host applications might be ordinary applications running on multiple network nodes, or smart contracts running on a blockchain. The *core layer* implements the cryptographic primitives, the protocols, and the orchestration logic. It uses the MIRACL Core library [10] internally, which provides elliptic curve cryptography, hash functions, and other cryptographic functions. The *network layer* provides abstractions for the communication between the peers within the network. It allows peers to exchange direct messages using the available network infrastructure, or to broadcast messages using a total order broadcast channel that is provided by the availability of an underlying blockchain.

Before this work, the keys used by the cryptographic primitives provided by Thetacrypt were generated using the trusted dealer approach (recall section 2.1.2). This work aims to add support for distributed key generation to Thetacrypt. Hence, we implement our DKG protocol as a part of Thetacrypt.

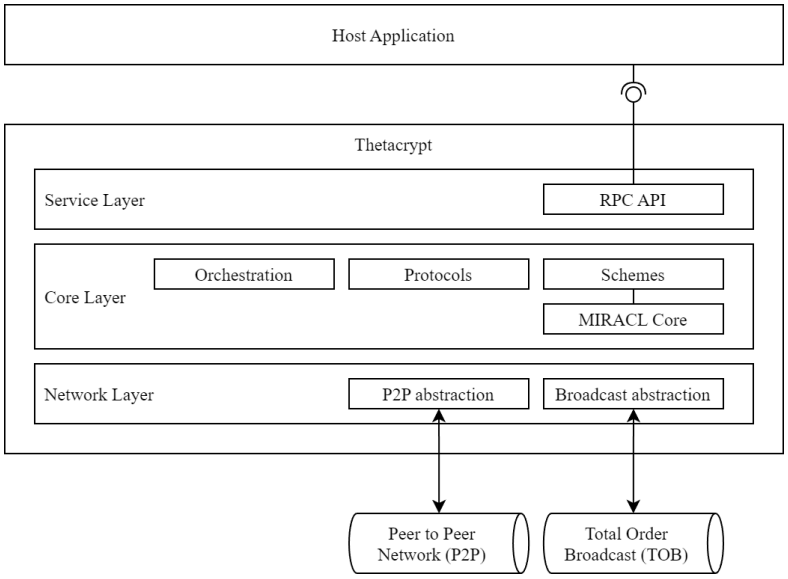

Figure 4.1. The architecture of Thetacrypt.

4.2 Protocol Architecture

Figure 4.2 shows the architecture of our implementation of the DKG protocol. We will elaborate on the responsibility and functionality of each component.

The *Thetacrypt node* is an instance of Thetacrypt running within a host application on some machine. The different instances are connected through a *peer-to-peer network* and a *total order broadcast channel* that allow them to exchange arbitrary messages. A Thetacrypt node may participate in multiple different protocol runs of different cryptographic operations at a time. For example, a node might participate in a decryption operation, while at the same time checking a signature on some document, and running a coin flip protocol in parallel. In figure 4.2, this is expressed with the component called *other protocol instances*. All protocols require the different nodes to exchange messages with each other. Hence, every node depends on a component called *instance manager* that decides for every incoming message to which protocol instance it belongs and dispatches it accordingly. It does so by checking a dedicated field in the message header that uniquely identifies a protocol instance. Note that before a message is handed over to the instance manager, there is some component that checks the integrity and authenticity of the incoming messages. We omitted this component in the figure to enhance readability.

So far, all components mentioned are independent of a specific protocol. This means that all dispatching decisions could be done without deserialization of the message payload or knowledge about the logic of the protocols. Once a message is passed to the correct protocol instance, knowledge about the protocol message types and internals is required to further process a message. There are two components responsible for

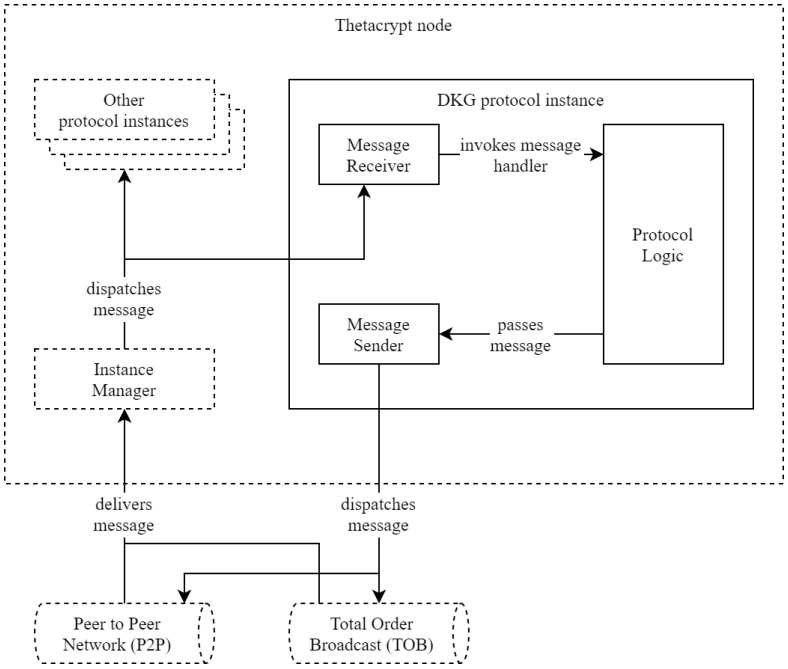

Figure 4.2. The architecture of our implementation of the DKG protocol. Components with a dashed border already existed before our work.

receiving and sending protocol-specific messages. Both are part of the protocol instance. The *message receiver* takes an incoming message, deserializes its payload, and passes it to the correct message handler of the *protocol logic implementation*. In our implementation, the message receiver just calls the correct function of the protocol logic, passing the deserialized message as an argument. The protocol logic then manipulates its state according to the rules of the protocol. As a consequence, messages to other parties might be sent, or a message might be broadcast to all parties. To do so, the protocol logic passes all necessary data to the *message sender*. That component serializes the data and publishes them either to the total order broadcast channel or sends them to some specific party through the peer-to-peer channel.

4.3 Implementation Details

In this section, we describe some specialties of the implementation that play an important role during the execution of the protocol, yet might not be that obvious when looking at the pseudocode in section 3.2.

4.3.1 Selection of Alternative Group Generators

Recall that in *GJKR-DKG* the existence of *two* generators g and h of the group G is assumed. The same holds for our protocol. Thetacrypt originally only provided a single generator per mathematical group it supports. Hence, we had to find additional group generators. The most important property of the two group generators is that the discrete logarithm of h with respect to g is not known. Hence, we chose a random group element. However, to ensure that an external party can verify that we created the generators the way we claim, we have to disclose the source of the randomness that we used to choose the group element.

To achieve this, we take some sequence of characters from a public source, hash it to a point on the elliptic curve on which the group is defined, and use that point as a generator. As a sequence of characters, we took the following string, encoded as UTF-8:

```
thetacrypt_6995e2de6891c724bfeb2db33d7b87775f913ad1
```

The first part is just the name of the Thetacrypt library, and the second part is the Git commit hash of version 6.4 of the Linux kernel [11]. We use the `map2point` function of the MiraclCore library to map the string to a point on the elliptic curve. Their implementation internally uses an SHA2 hash function, and then follows the approach described in [12] to map the hashed input to a point on the elliptic curve.

We created the generators once and defined them in the source code as constants for further use. This gives better performance compared to recreating them whenever Thetacrypt is initialized. The script that was used for generator creation is included in the Thetacrypt. This allows users of the service to verify that the constants defined in the code match the ones that are output by the script.

4.3.2 Deterministic Ordering of Protocol Parties

Both *GJKR-DKG* and our own protocol assume the existence of some deterministic global ordering of the parties participating in the protocol. This ordering is used at multiple places in the protocol:

- When creating shares.
- When verifying shares in equation 2.1.
- When building the verification vector and the verification matrix for assigning a row and a column to a party.
- When selecting $2t + 1$ parties out of the candidates to form *QUAL*.
- When verifying commitments in equation 2.2.

There are multiple solutions to obtain this ordering. We suggest taking the binary value of the IP address of each party, ordering those in ascending order, and using this ordering in the protocol. Note that this ordering breaks if a new party joins the network. However, since Thetacrypt does not support dynamic membership at the moment, this is a usable method.

4.3.3 Construction of the Qualified Set

Recall that, during the sharing phase, we have to decide on the set *QUAL* of parties whose input will be used for the computation of the secret and the public key. We do so by finding a subset of $2t + 1$ out of the n parties who received valid shares from all parties in that subset. In our implementation, we use a naïve approach to find the subset. We simply enumerate the possible candidate sets in a deterministic way and check whether the condition holds for the members of that candidate set.

This approach is computationally expensive for large n since the number of possible candidate sets is given by $\binom{n}{2t+1}$. Hence, the enumeration of all candidate sets is an exponential time operation. Finding a more optimized way to construct the candidate set is an optimization left to future work.

Note that it is critical for the protocol that the enumeration of the candidate sets produces them in the same order for all (honest) parties. Otherwise, there is no guarantee that all honest parties end up with the same set *QUAL*. In our implementation, we consider the parties according to their global ordering and then use the algorithm provided by Itertools [13] to generate the candidate sets.

4.3.4 Commitment Schemes

Our protocol makes use of two different commitment schemes. During the sharing phase, it uses a *Pedersen Commitment* [3] to commit to the coefficients $a_{i0}, \ldots, a_{it}$ and $b_{i0}, \ldots, b_{it}$ of the sharing polynomials f_i and f'_i. The commitment is computed as $g^{a_{ik}} h^{b_{ik}} \mod p$ for $k = 1, \ldots, t$ and is information-theoretically hiding.

During the public key reconstruction phase, our protocol uses a simple *discrete log*-based commitment to commit to the coefficients $a_{i0}, \ldots, a_{it}$ of the sharing polynomial f_i. In this case, the commitment is computed as $g^{a_{ik}} \mod p$ for $k = 1, \ldots, t$ and is computationally hiding.

We decided to implement both commitment schemes in Thetacrypt as a separate module such that they can be reused by other protocols in Thetacrypt. Listing 1 shows an example of how the commitment schemes (Pedersen in this example) can be used in Thetacrypt.

```
let group = Group::Bls12381;
let mut rng = RNG::new(RngAlgorithm::MarsagliaZaman);

// Define the value we want to commit to.
let x = BigImpl::new_rand(&group, &group.get_order(), &mut rng);

// Get some randomness that is used for hiding.
let r = BigImpl::new_rand(&group, &group.get_order(), &mut rng);

let params = CommitmentParams::Pedersen(
                PedersenCommitmentParams::init(x, r));

// Commit to x, using the randomness r for hiding.
let c = Commitment::commit(&params);

// Verify whether the commitment holds. The function would
// return false if we pass a different x or r to it.
let result = c.verify(&params);

assert(result == true);
```

Listing 1. Usage of the Pedersen Commitment implementation.

4.4 Testing

The source code of Thetacrypt contains a set of unit tests that verify the correctness of its functionality. However, not all parts of the source code are equally well covered by unit tests. Furthermore, the usual process to test some functionality of Thetacrypt is to bring up multiple instances of Thetacrypt using a local deployment and initialize a protocol run. This process is cumbersome and slow and becomes more complex if a blockchain platform is used underneath. We were looking for an easier and faster way to test the DKG protocol.

We followed two approaches for testing that are in line with the well-known testing pyramid, as shown in figure 4.3. First, we use *unit tests* to cover the individual compo-

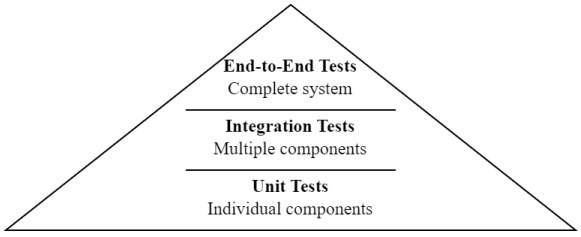

Figure 4.3. The test pyramid, showing the different test levels and their scope.

nents of our implementation. This means that we mock the behavior of dependencies of a certain component to test a component in isolation. Second, we use *integration tests* to test the integration of the different DKG-related components. In our case, this means that we simulate the network and run multiple participating parties in different threads of a single test process.

4.4.1 Unit Testing

To facilitate unit testing, we aimed to decompose the protocol implementation into smaller components with well-defined responsibilities that can be individually tested. Parts of this approach are already shown in figure 4.2 when we elaborated on the architecture of our implementation and the role of the message sender and receiver. But the decomposition is also present on a lower level of the code. For instance, we introduced a dedicated struct *VerificationTable* that represents the verification matrix, along with the operations that manipulate it. Remember that two instances of a verification matrix are used in our protocol: one during the sharing phase, and one during the public key reconstruction phase. Factoring that functionality out into a dedicated struct allows code reuse and reduces redundancy. We also use a dedicated component that is responsible for sending messages to other parties. Its responsibility is to assemble and serialize messages and to post them to the right communication channel. Hence, the protocol logic is not cluttered with message serialization and message dispatching code.

4.4.2 Integration Testing

To implement our integration testing approach we had to provide mock implementations for the message sender and receiver that underneath simulate the behavior of the network. Hence, we had to find or implement replacements for the peer-to-peer channels and the total order broadcast channel that work in-process and allow message exchange between different threads. Fortunately, the well-known Rust library *Tokio* [14] that is already used in Thetacrypt provides primitives for both:

- The `tokio::sync::broadcast` structs and functions provide a broadcast implementation for inter-process communication that guarantees the ordering of messages.
- The `tokio:sync::mpsc` structs and functions provide a multi-producer-single-consumer channel implementation for inter-process communication.

The listings 2 - 6 show an integration test of our protocol implementation. The test is divided into a *setup*, a *running*, and a *verification* stage. During the setup stage, the test first defines the global parameters like the mathematical group G, including its generators g and h. Furthermore, it chooses n and t and generates a set of party identifiers.

```
// Define a set of party IDs and their unique index
let party_ids: BTreeMap<String, usize> = create_party_ids();
//Set up global parameters like the mathematical group, n, and t.
let params = setup_global_parameters(&party_ids);
```

Listing 2. Protocol integration test, setup stage: global parameters

In the second step of the setup stage, the test creates all the communication channels, the parties participating in the protocol themselves, along with some supporting components.

```
// Set up broadcast channel.
let (broadcast_sender, _broadcast_receiver) =
    tokio::sync::broadcast::channel(100);

// Set up the channel through which the parties will expose
// some results of their local computations for testing purposes.
let (result_sender, result_receiver) =
    tokio::sync::mpsc::channel(100);

// Set up channel for p2p broker.
let (p2p_broker_sender, p2p_broker_receiver) =
    tokio::sync::mpsc::channel(100);

// Create the parties participating in the protocol, along with
// their communication channels.
let (parties, party_p2p_senders) = create_parties(
    &party_ids, &params, p2p_broker_sender, &broadcast_sender,
    result_sender);

// Set up the broker that will dispatch direct messages
let mut p2p_broker = create_p2p_broker(
    p2p_broker_receiver, party_p2p_senders);

// Set up the result collector who gathers all protocol results
// from all parties.
let mut result_collector = create_result_collector(
    result_receiver);
```

Listing 3. Protocol integration test, setup stage: components

Note that we do not use a dedicated channel to bidirectionally connect every pair of parties. Instead, we use a central message broker. If a party wants to send a direct message to some other party, it sends the message to the central broker, which then forwards it to the other party. This allows us to reduce the number of point-to-point communication channels from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ but introduces a central instance for communication dispatching. In the real system, this would not be tolerable, but in our testing scenario, this is sufficient as the security aspect is not of interest. Furthermore, we can manipulate the message exchange at that central point to simulate an adversary that,

for instance, prevents messages from a certain party from being delivered. This enables us to verify that the protocol works correctly despite adversarial actions.

Furthermore, we let each party p_i expose some of its results to a so-called result collector. These results include the chosen secret x_i, the selected set *QUAL*, as well as the secret key share and the public key y it computed. Of course, no party would expose those values in a real deployment. We only expose those values in the test to be able to verify the correctness of the protocol run at the end of the test.

The last step during the setup stage is to create and start all the threads that will run a component each.

```
36 let mut threads = Vec::new();
37
38 // Set up the threads that run one party each.
39 for mut party in parties {
40     let party_thread = spawn(async move {
41         party.listen().await;
42     });
43
44     threads.push(party_thread);
45 }
46
47 // Set up the thread that runs the message broker.
48 let p2p_broker_thread = spawn(async move {
49     p2p_broker.listen().await;
50 });
51
52 threads.push(p2p_broker_thread);
53
54 // Set up the thread that runs the result collector.
55 let result_collector_thread: JoinHandle<Vec<DkgResultMessage>> =
56     spawn(async move {
57         let results = result_collector.listen().await;
58         return results;
59     });
```

Listing 4. Protocol integration test, setup stage: threads creation

In the running stage we trigger a protocol run by broadcasting a `Run` message to all parties. The parties will then run the protocol. We wait for all parties to finish.

```
61 // Run protocol
62 let _result = broadcast_sender.send(DkgInboundMessage::Run);
63
64 // Wait for protocol to finish
65 for thread in threads {
66     let _thread_join_result = thread.await;
67 }
```

Listing 5. Protocol integration test, running stage: run protocol and wait for completion

During the verification stage, we first gather all the results that the result collector received from the different parties. Finally, we evaluate a set of assertions against the collected results to validate that the parties computed the expected results:

- We validate that all (correct) parties computed the same set *QUAL*.
- We validate that all (correct) parties in *QUAL* computed the same public key y.
- We validate that the public key y corresponds to the secrets z_i that were chosen by the qualified parties at the beginning of the protocol. That is we verify that $y = g^{\sum_{p_i \in QUAL} z_i}$
- We validate that the secret key shares x_i computed by the parties in *QUAL* can be used to reconstruct the secret key x, and it holds that $y = g^x$.

```
69 // Collect the result of the protocol run
70 let results = result_collector_thread.await.unwrap();
71
72 let chosen_secrets = get_chosen_secrets(&results);
73 let qualified_parties = get_computed_qualified_sets(&results);
74 let public_keys = get_computed_public_keys(&results);
75 let secret_key_shares = get_computed_secret_key_shares(&results);
76
77 // Assert that all parties computed the same qualified set.
78 let qualified_parties =
79     assert_unique_qualified_parties(&qualified_parties);
80
81 // Assert that all parties computed the same public key.
82 let public_key = assert_unique_public_key(&public_keys);
83
84 // Assert that the public key corresponds to the secrets chosen
85 // by the qualified parties.
86 assert_public_key_corresponds_to_chosen_secrets(
87     &public_key, &chosen_secrets, &qualified_parties);
88
89 // Assert that the public key corresponds to the secret key
90 // defined by the secret key shares.
91 assert_public_key_corresponds_to_secret_key(
92     &public_key, &secret_key_shares, &qualified_parties);
```

Listing 6. Protocol integration test, verifying stage

5 Conclusion

5.1 Our Contribution

In this work, we took the existing synchronous DKG protocol of Gennaro *et al.* and transformed it into an asynchronous protocol. Our protocol is designed in a modular way. It exploits the properties of a total order broadcast primitive and authenticated point-to-point channels, which makes it usable on top of existing environments like blockchains that provide an implementation of those primitives.

Our protocol provides security against an adversary that corrupts up to $t < n/3$ parties. The protocol has the same simplicity as the original synchronous protocol. It consists of a sharing phase and a public key reconstruction phase. During the first phase, the parties decide on a set of parties *QUAL* whose inputs will be used to generate a secret key. They do so by building and broadcasting so-called verification vectors to vote on whether or not a certain party behaved correctly. The properties of the total order broadcast ensure that all correct parties end up with the same set *QUAL*. During the second phase, the parties in *QUAL* collaboratively reconstruct the public key based on their inputs. They again hold a vote on the parties in *QUAL* that behaved correctly, reconstructing the inputs of the parties that showed malicious behavior.

We implemented our protocol in the Rust programming language and integrated it into the Thetacrypt threshold cryptography service. Furthermore, we developed a testing infrastructure that allows running the protocol on a single machine within a single process, using multiple threads to simulate the different parties participating in the protocol. This enables testing of the protocol and facilitates further development.

This approach can also be generalized to the other protocols and schemes in Thetacrypt, resulting in a better developing and testing experience.

5.2 Future Work

The treatment of certain aspects of our protocol exceeded the scope of this work, and, hence, is left for future work. First, our protocol lacks formal correctness and security proof. Coming up with such proofs should not be hard as one can build on the proofs of the original synchronous *GJKR-DKG* protocol. Second, the Thetacrypt service does not yet support open membership. That is, the number of parties is fixed upon initialization of the system and cannot be changed later. Using our DKG protocol makes it possible to add the open membership feature to Thetacrypt in the future. Third, we use the simplest algorithm one can think of to find the qualified set by simply enumerating possible candidate sets, which is a $\mathcal{O}(2^n)$ operation. One should aim at optimizing this part of the protocol. Fourth, we did not assess the performance and scalability of our protocol. It would be interesting to run experiments on a real deployment of the system to obtain reliable performance metrics for our protocol and compare it to others.

6
Bibliography

[1] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *J. Cryptology*, vol. 20, pp. 51–83, 2007.

[2] cryptobern, "Thetacrypt." GitHub. `https://github.com/cryptobern/thetacrypt`, retrieved 2023-10-16.

[3] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology — CRYPTO '91*, (Berlin, Heidelberg), pp. 129–140, Springer Berlin Heidelberg, 1992.

[4] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures.," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, (New York, NY, USA), p. 1751–1767, Association for Computing Machinery, 2020.

[5] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Reaching consensus for asynchronous distributed key generation," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, (New York, NY, USA), p. 363–373, Association for Computing Machinery, 2021.

[6] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, p. 612–613, nov 1979.

[7] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[8] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 2518–2534, 2022.

[9] "Cryptology and data security research group." Online. `https://crypto.unibe.ch/`, retrieved 2023-10-16.

[10] Miracl, "Miracl/core: Miracl core crypto library." GitHub. `https://github.com/miracl/core`, retrieved 2023-10-16.

[11] L. Torvalds, "Linux 6.4." GitHub. `https://github.com/torvalds/linux/commit/6995e2d`, retrieved 2023-10-16.

[12] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood, "Hashing to elliptic curves," Internet-Draft draft-irtf-cfrg-hash-to-curve-06, IETF Secretariat, March 2020.

[13] "Itertools, k-combinations iterator." Online. `https://docs.rs/itertools/latest/itertools/trait.Itertools.html#method.combinations`, retrieved 2023-12-04.

[14] "Tokio, an asynchronous runtime for rust." Online. `https://tokio.rs/`, retrieved 2023-10-23.

Declaration of consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name: Eggimann Markus

Registration Number: 15-101-959

Study program: MSc Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Title of the thesis: Design and Implementation of an Asynchronous Distributed Key Generation Protocol

Supervisor: Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.

For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Ruswil, 30.10.2023

Place/Date

M. Eggimann

Signature