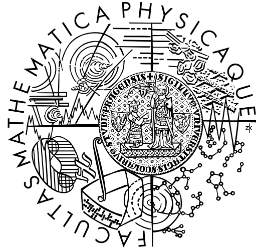


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

František Mejzlík

Fast hash-based signing protocol for message stream authentication

Department of Software Engineering

Supervisor of the master thesis: RNDr. Filip Zavoral, Ph.D.
Study programme: Computer Science
Study branch: Software Systems

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In . . . . date . . . .

Author's signature

Herby, I thank the thesis supervisor RNDr. Filip Zavoral, Ph.D., as well as the thesis consultant RNDr. Miroslav Kratochvíl, Ph.D., for the invaluable guidance and critical feedback they have patiently provided me throughout the process. Moreover, I thank my dear parents and my family for all they have been giving to me (and taking away from me) throughout my whole life.

Title: Fast hash-based signing protocol for message stream authentication

Author: František Mejzlík

Department: Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D., Department of Software Engineering

Abstract: Security of the data streaming over Internet becomes a challenge if requirements such as post-quantum-capable cryptography and complete decentralisation must be addressed. This thesis develops a connection-less, re-broadcastable data streaming protocol that allows a wholly decentralised, petname-based quantum-robust authentication of streaming sources based solely on the post-quantum hash-based few-time signature schemes. As the main contribution, the thesis benchmarks various trade-offs given by the problematic ephemeral nature of identities based on the few-time signature schemes and by the desired networking properties of the streaming protocol. The benchmarks show that the schemes are practically extensible to realistic use cases, with only minor overhead. The proof-of-concept protocol implementation is provided as a Rust library, together with the example application for live audio broadcasting.

Keywords: message stream post-quantum authentication protocol hash-based signatures

Contents

Introduction **3**

1 Hash-based signatures **7**

1.1 The motivation for hash-based cryptography 7
1.2 Basic terminology 8
1.2.1 Cryptographic hash functions 8
1.2.2 Random number generators 11
1.3 Constructing signatures from hash functions 12
1.3.1 One-time signature schemes 13
1.3.2 Few-time signature schemes 17
1.3.3 Current development in hash-based signatures 24

2 Authenticated streaming with hash-based signatures **27**

2.1 Design Criteria 27
2.1.1 Purpose 28
2.1.2 Scope 28
2.1.3 Main abstractions 30
2.1.4 Requirements 31
2.1.5 The interface of the protocol 32
2.2 Specification of the protocol 33
2.2.1 Network formats 35
2.2.2 Sender role 37
2.2.3 Receiver role 45
2.2.4 Distributor role 55
2.3 Parameter choice trade-offs 55
2.3.1 Security vs. signature size 57
2.3.2 Re-authenticating the prior sender identities 58

3 Proof-of-concept implementation **61**

3.1 HAB: A library for Hash-based Authentication Broadcasting . . 62
3.1.1 Overview of the HAB interface 62

3.1.2 Internal structure of HAB library 67
3.2 AudiBro: A real-time audio broadcasting 74
3.2.1 Interacting with the application 75
3.2.2 Design of the AudiBro application 75

4 Results and performance **79**
4.1 Finding practical protocol parameters 79
4.2 Parametrising the re-authentication delay 81
4.2.1 Measuring methodology 84
4.3 Data overhead trade-offs 84
4.4 Protocol behaviour with practical parameters 87

Conclusion **91**

Bibliography **93**

A Structure of the attached software **97**
A.1 `hab` 97
A.2 `audibro` 97

B Using the HAB library **99**
B.1 Sender 100
B.2 Receiver 100

C Using the AudiBro application **101**

Introduction

In the current interconnected world, data streaming, multicasting and broadcasting over networks have become part of everyday life for many people. Consequently, high volumes of data go through the insecure Internet, and often such data are sensitive and need some verification. Unfortunately, secure data streaming and broadcasting continue to face significant challenges. These challenges are particularly prominent in ensuring the integrity and authentication of data sources, especially with real-time data. In other words, to know who sent the data upon their receiving. With advances in quantum computing and as the volume of data swarming in networks rapidly increases, the need for robust and performant security measures is becoming more crucial by the day. A vital aspect of this endeavour is to develop data authentication and integrity mechanisms for real-time data streaming and broadcasting that keep up with technological advancements and can withstand future threats.

Nowadays, some commonly used protocols for data streaming [1, 2] leverage standardised asymmetric cryptography — for example, Digital Signature Algorithm (DSA) [3], Rivest-Shamir-Adleman (RSA) [4] or Elliptic Curve Cryptography (ECC) [5]. Some protocols, like TESLA [6], are built upon symmetric cryptography — for example, using Message Authenticating Codes (MAC). There is also the BiBa signature scheme with its streaming protocol [7] based on *hash-based cryptography* — more specifically, it employs *few-time signature* (FTS) schemes.

However, commonly used protocols usually do not provide quantum-resistant authentication. Additionally, due to their connection-oriented nature, they are not suitable for distributed data dissemination. Therefore, these solutions are insufficient for future advances in quantum computing.

Previous work The previously mentioned symmetric cryptography solution, the TESLA protocol, is based on MACs and can be constructed to be quantum-resistant. On the other hand, the protocol assumes the loose time synchronisation of the sender and receiver; this means that if the sender and receiver differ in time more than some specific value, the authenticity cannot be guaranteed.

The streaming protocol proposed alongside the BiBa signature scheme is

post-quantum but suffers some drawbacks. Most importantly, it is bound only to the specific signature scheme that can be found unsatisfactory in the future. The protocol cannot be trivially adjusted to use some other scheme. Also, since the protocol is inspired by the TESLA protocol, it requires loose time synchronisation, just as the TESLA protocol. Finally, the protocol is not robust enough for huge packet losses. For example, such losses can be caused by a receiver going offline for a few days or weeks and then reconnecting again.

Goals of the thesis The primary objective of this thesis is to evaluate the suitability of hash-based signatures for the authentication of data streams, with a particular emphasis on real-time applications. Hence, it proposes a protocol, investigates the overhead, and examines the influence of configuration parameters that optimise the protocol for specific use cases.

This thesis proposes a general protocol for authenticated data streaming that uses identities with petnames and relies on hash-based few-time signature schemes. Designed for real-time or non-real-time applications where reliability is not required, the protocol offers both post-quantum authentication and integrity of the data as well as identity management of known senders. With a focus on practicality, the proposed solution provides tolerance for significant packet losses — such as when a connection is unavailable for prolonged periods — enabling subsequent re-authentication of data coming from an already known sender and the possibility for distributed re-sending of the messages to load balance. This data re-distribution can be an alternative to routed multicast in IPv6 that is not yet usable on the Internet; the routed IPv6 multicast could save a lot of bandwidth. As a weaker alternative, instead of streaming data peer-to-peer to each receiver from one source, we can send the data to fewer receivers that will re-distribute the data to topologically closer receivers. A similar concept is used with offline data in Content Delivery Networks (CDNs) or in Google Global Cache [1], where the ISPs locally serve the cached data of Google.

The protocol itself is independent of the underlying signature scheme as long as it is post-quantum; of course, the protocol would work with a scheme that is not quantum-resistant but only with the security the scheme provides. Moreover, the proposed protocol is highly adjustable by parameters that make it possible to fit the configuration for various different usages. With these parameters, one can balance, for example, the overhead size versus security. This thesis also presents a highly configurable proof-of-concept implementation library with a HORST (section 1.3.2) signature scheme and displays recommended configurations for real-time data streaming with different security requirements. Also, it visualises the effect of parameters on protocol security and its overhead. On top of that,

[1] `https://support.google.com/interconnect/answer/9058809`

it shows the expected number of messages that can be lost without losing the ability to re-authenticate known identities. Furthermore, the example usage of this library is demonstrated on the simple application designed for live audio broadcasting and secure receiving.

This thesis is structured as follows: The first chapter introduces and describes the essential building blocks for constructing the proposed protocol. Such building blocks are *cryptographic hash functions*, *pseudo-random number generators* or one-time and few-time signature schemes. The introduction of fundamental primitives is followed by the chapter specifying the design criteria and the protocol. With the protocol knowledge, the thesis then thoroughly describes the proof-of-concept implementation of the proposed protocol in the form of a library. Moreover, the library implementation is complemented by an application for live audio streaming, demonstrating how the library can be used. Then this thesis delves into practical configurations, examining the protocol performance and anticipated behaviour in real-world scenarios. Finally, the conclusion and proposals for future work are presented.

Chapter 1

Hash-based signatures

The first chapter of this thesis provides the motivation, background, and context. The chapter starts with discussing what post-quantum means and why it is vital to work on such alternatives; it continues with introducing the essential cryptographic primitives — cryptographic hash functions and pseudo-random generators. This chapter concludes with a description of one-time and few-time signature schemes, focusing on the HORST scheme used as a foundation for the proof-of-concept implementation described in the chapter 3.

1.1 The motivation for hash-based cryptography

Before introducing the main used cryptographical primitives, it is crucial to explain why post-quantum cryptography matters and why it is essential to develop also hash-based cryptographic solutions.

Hash-based cryptography is one of the directions of post-quantum cryptography (PQC). PQC refers to algorithms and protocols that are designed to be secure against attacks leveraging quantum computers. The widely-used asymmetric cryptography algorithms usually rely on three computationally hard problems — integer factorisation, discrete logarithm and elliptic-curve logarithm problem. All these can be solved easily with a powerful enough quantum computer running Shor's algorithm [8]. Even though the current quantum computers are not sufficiently powerful yet, it is important to develop alternatives for when this will be the case. Post-quantum cryptography has gained a lot of attention lately, and many people and organisations are working on quantum-resistant cryptography solutions. Even the National Institute of Standards and Technology (NIST) has started the Post-quantum Cryptography Standardization project in 2016 [1]. The project aims to compile, analyse and standardise the algorithms that

[1]`https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/`

can withstand quantum computer attacks.

The domain of hash-based cryptography is mainly about constructing secure digital signatures as an alternative to standard algorithms; an example of such an algorithm is DSA [3], which is based on the complexity of the discrete logarithm problem. Hash-based cryptography algorithms are among those whose security relies on much weaker assumptions — these were described in section 1.2.1. As the name suggests, they are based on cryptographic hash functions because there exist functions that are considered secure in the critical security aspects described earlier in this section. Other directions of post-quantum cryptography include lattice-based cryptography [9], multivariate cryptography [10], code-based cryptography [11] or supersingular elliptic curve isogeny cryptography [12].

1.2 Basic terminology

As this thesis focuses on designing a post-quantum protocol for authenticated broadcasting, it is crucial to understand the underlying cryptographic primitives upon which the protocol is built. Firstly, an introduction to the critical components of the protocol is presented, including cryptographic hash functions, random number generators, and the concept of post-quantum cryptography. The subchapter also highlights the significance of developing post-quantum alternatives to traditional cryptography, emphasising the need for secure communication in the face of future advancements in quantum computing.

1.2.1 Cryptographic hash functions

Before introducing cryptographic hash functions, it is vital to understand the concept of one-way functions. The notion of one-way function has been used in constructions since the eighties [13, 14]. They were also called 'one-way cyphers' as an analogy to standard reversible cyphers. Intuitively, a one-way function is a mathematical function with the extra property of 'one-wayness'. This additional property requires the function to be easy to compute the output (also referred to as image) for all possible inputs (also referred to as pre-images), but it should be computationally infeasible to revert the process; that is, for the given image to find what was the pre-image that led to this specific image. One-way functions or their construction, however, still lack a formal proof of existence. Despite that, many proposed functions appear to behave like one-way functions and are used at places where the theoretical constructions assume one-way functions.

Definition 1 (One-way function)**.** *A function $f : X \to Y$ is called one-way function if $f(x)$ is easy to compute for all $x \in X$ and is computationally infeasible to to find any $x \in X$ for a random $y \in Y$ such that $f(x) = y$.*

This definition is not the most rigorous but sketches the intuition and meaning behind it. The words 'easy to compute' refer to a lower-degree polynomial algorithm — the lower, the better. Consequently, 'computationally infeasible' refers to a time or space complexity of a computation that no computer — nor cluster of computers — can compute in an economically feasible way — in this case, the higher the complexity, the better.

Hash functions Generally, a hash function is a mathematical function that maps the arbitrarily large space of pre-images (a function domain) to a set of fixed-size images (a function codomain). Usually, hash functions operate on binary strings — sequences consisting of ones and zeroes. This output is often called a digest or a fingerprint in the context of hash functions. An example of such a function is a parity check. It operates on arbitrary input of ones and zeroes; the space of images of such a function contains only two images — zero and one. The codomain is a set of two elements of size one bit. Although this example is rather simple, hash functions are essential in cryptographic use cases for reducing the size of inputs that the subsequent algorithms need to process; these are reduced to fixed-size inputs. Also, hash functions are popular because they are believed to be hard to break. Naturally, there are multiple types of hash functions; different types are designed to serve different purposes and therefore are put different requirements on them. Example types are checksum or non-cryptographic functions; these are not intended for cryptography but for other purposes, which may include fash hashing for key generation in hash tables, for example.

The term 'hash function' is often used in the sense of a hash function with some specific properties rather than just a function that only reduces the space to fixed-size elements. For the rest of this thesis, the term hash function refers to cryptographic hash functions unless explicitly stated otherwise.

Cryptographic hash functions A cryptographic hash function is a combination of a one-way and hash function. Is it also often called a one-way hash function, even though there aren't any strictly-followed rules in naming. The important thing here is what is meant by this term. A cryptographic hash function is a computationally difficult function to invert that maps arbitrarily long binary string input into fixed-size binary output and has the following properties. The term 'invert' does not refer to a mathematical inversion. Let's denote the cryptographic function as h.

Pre-image resistance For the given output — the image, it must be computationally infeasible to find any function input — the pre-image — that would

map to the same image. In other words, for the output y, find x such that $h(x) = y$.

Collision resistance Finding two distinct pre-images that map to the same image must be computationally infeasible. To put it formally, to find x and x', $x \neq x'$ such that $h(x) = h(x')$.

Second pre-image resistance For a given pre-image, it should be computationally infeasible to find a different pre-image that maps to the same image. In other words, for given x to, find x' and $x \neq x'$ such that $h(x) = h(x')$.

These properties give a high level of confidence that it is too computationally demanding for an adversary to modify the function input such that it produces the same output. Therefore, we can assume that the two inputs with the same output are — with sufficiently high probability, at least — identical. Also, collision resistance implies second pre-image resistance because if one has found a second pre-image for some input, one has just found a collision too.

Bit security interpretation of attack complexities Although, to the author's best knowledge, there is no formal proof that the specific cryptographic hash function has the three beforementioned security properties (which is inherited from the lack of proof that would imply the existence of one-way functions), these properties can be proven with a high degree of confidence through extensive analysis testing and construction of attacks against these functions. Hash functions are considered *unbroken* if these specialised attacks against the concrete hash function construction are at most as effective as generic attacks. Generic attacks do not know any information about the construction, nor do they exploit any vulnerabilities or weaknesses; these use the function as a black box and rely only on general hash function properties. An example of such an attack is a birthday attack [15] that implies that the collision resistance of any hash function is at most $2^{\frac{n}{2}}$ where n is the bit size of the output.

Provided that h is a perfect and unbroken cryptographic hash function with an output size of n, we can declare the function security against the attack to the beforementioned properties. First, let's take a look at how it looks with the standard computer [16]. The classical computer would need to do $O(2^n)$ hash function invocations to break pre-image or second pre-image resistance; thus bit security of these properties is n. As for collision resistance, the number is significantly lower due to the birthday paradox and birthday attack against the hash function collisions; only $O(2^{\frac{n}{2}})$ invocations are needed. That means that the bit security of the hash function with an output size of n is $\frac{n}{2}$.

Regarding quantum computers, the most problematic part is pre-image resistance [17]. Due to Grover algorithm [18], it is possible to find pre-image in only $O(2^{\frac{n}{2}})$ time. Additionally, [19] states that a quantum computer can find a collision in $O(2^{\frac{n}{3}})$ time. Bernstein later disputed this statement [17], pointing out that the algorithm indeed runs in $O(2^{\frac{n}{3}})$ time but also requires a quantum computer of size $O(2^{\frac{n}{3}})$ at which scale even speedup of a classical computer can be higher. We can see that even with the worst-case scenario, with the potential ability to find collisions in $O(2^{\frac{n}{3}})$ time, the security of hash functions is not destroyed. One can always use a hash function with a larger output size to compensate for the security lost due to quantum attacks.

Ultimately, the real-world implementations of cryptographic hash functions try to be as close as possible to this ideal hash function. If one could implement such a function, there would be no better attack than the generic one against that function. Then the security of cryptographic algorithms built on that hash function would boil down to the security of a perfect hash function.

1.2.2 Random number generators

Cryptography would make no sense if there were no way of generating secret keys unpredictably for an adversary. Thus, random number generators are one of the most fundamental cryptography primitives. In the context of this thesis, when speaking about a random bit or random number generator (RNG), it is meant for an algorithm that produces sequences of statistically independent binary digits — bits. From the bits, to form a number of a given bit length n, it is required to generate, in total, $log_2 n$ bits that represent the number.

Definition 2 (Random bit generator)**.** *A random bit generator is an algorithm or device that produces a sequence of statistically independent bits.*

Such primitives are also called true random number generators (True RNGs) because their output is — at least in theory — unpredictable. Additionally, there are also pseudo-random number generators (PRNGs), which are also of significant importance in cryptography. Pseudo-random number generators are deterministic algorithms taking an input called seed and producing much larger random bit sequences that appear to be random. In fact, the outputs are not random because whenever one seeds the same PRNG algorithm with the same seed, the same number sequence is yielded.

Definition 3 (Pseudo-random bit generator)**.** *A pseudo-random bit generator is a deterministic algorithm that produces a bit sequence that appears random based on the provided seed. The size of the output sequence should be significantly larger than s.*

The importance of PRNGs is that their real-world implementations are purely software solutions and do not rely on any devices that serve as a source of entropy for the true number generation (e.g. `/dev/random` device on Linux systems). Thus, they can produce pseudo-random numbers much faster. Moreover, it is often necessary to reproduce the existing results, and therefore one needs a way to generate the same 'random' numbers as in the first original run.

In the previous definition, the term 'bit sequence that appears random' was used. A good PRNG to be used as a foundation for security should provide some quality of its outputs. The minimum security requirement for a PRNG is that the bit size of the seed should be sufficiently large, so it is infeasible for an adversary to iterate over possible seed values.

Moreover, the bit outputs should be statistically indistinguishable from true random bit sequences. The successful output prediction should be computationally hard. A common property that the PRNGs must satisfy is passing the next-bit test.

Definition 4 (Next-bit-test)**.** *Let y be the output pseudo-random sequence. A pseudo-random bit generator passes the next-bit test if there is no polynomial-time algorithm which, given the first k bits of y, can predict the $k+1$-th bit of y with a probability significantly greater than $\frac{1}{2}$.*

The PRNG that passes the next-bit test is said to be a cryptographically secure pseudo-random number generator (CS PRNG).

1.3 Constructing signatures from hash functions

The idea of constructing a digital signature based on a one-way function was first mentioned in 1976 inside the journal New Directions in Cryptography in [20]. This publication was followed by Rabin [21], who described a working yet impractical signature scheme where parts of the secret key needed to be present for the signature verification, thus requiring the cooperation of the signed and verifier. Finally, Lamport extended the solution to a digital signature algorithm known as Lamport or Lamport-Diffie one-time signature scheme [22]. Lamport's signature is the scheme that this thesis begins with on the route toward constructing the quantum-resistant broadcast authentication protocol based on cryptographic hash functions. Although the scheme that Lamport proposed is not the most practical, it let other authors iteratively build on that idea and develop more practical schemes; from the eighties through now, where robust signature schemes like SPHINCS+ [23] have been proposed and are being standardised.

1.3.1 One-time signature schemes

One crucial property of the Lamport signature scheme is that one key pair can securely sign only one message; schemes like this fall into the category of one-time signature schemes (OTS). One-time signatures are often essential inner building blocks of more complex schemes. Moreover, there are ways to implement a few-time signature scheme (FTS) from those that are one-time; these schemes can then securely sign a larger, although not infinite, number of messages.

Lamport one-time signature scheme

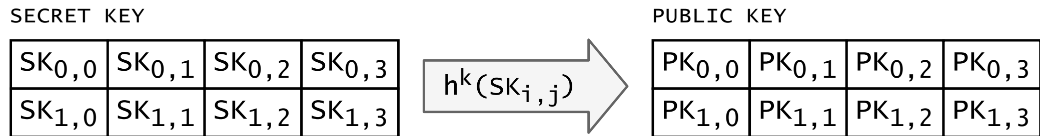
The Lamport signature scheme [22] relies on a one-way function and a secure seeded pseudo-random number generator. Usually, in descriptions, one-way functions are substituted with cryptographic hash functions for convenience — since they are also one-way. The scheme works as sketched in the figure 1.1; it starts with a randomly generated secret key consisting of two sets of numbers and hash values of these as a public key. The main idea is that the signature includes the sequence of numbers from the secret key, either from the first or second set of the keys, based on the bit value currently being signed. Verifying the signature starts with hashing the message; the verification is done in the same way as for signing. Then it processes the individual bits of the digest, hashes the corresponding number from the signature and checks that it matches the one in the public key. The signature is considered valid if this holds for all bits in the message digest; in this scheme, each secret number from the secret key signs one bit of the digest.

Primitives The scheme uses a seeded cryptographically safe pseudo-random number generator for secret key generation and a one-way function $h^k : \{0,1\}^n \to \{0,1\}^n$ for public key derivation and verification. Also, it uses a cryptographic hash function $h^m : \{0,1\}^* \to \{0,1\}^n$ for producing a digest of a message m. For convenience, the same cryptographic hash function h is usually used in real-world implementation in both roles.

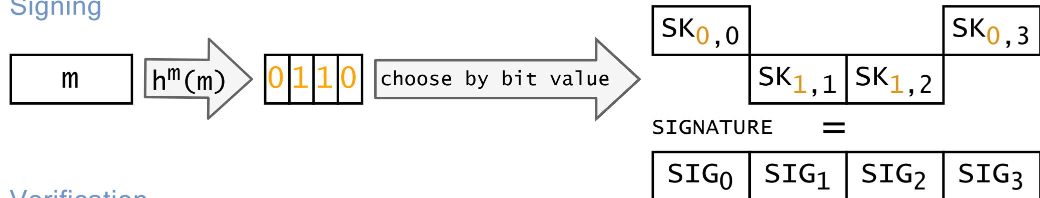
Parameters The security parameter $n \in \mathbb{N}$. This parameter sets the output sizes for the function h^k, h^m and denotes how many random numbers form a secret key — and thus also a public key.

Key generation Using the PRNG, generate $2n$ random numbers of size n; this is the secret key $\mathrm{SK} = (\mathrm{SK}_0, ..., \mathrm{SK}_{2n})$. Compute the public key by hashing the numbers that are part of the secret key; do so with h^k; i.e. $\mathrm{PK} = (h^k(\mathrm{SK}_0), .., h^k(\mathrm{SK}_{2n}))$.

Key generation



Signing



Verification

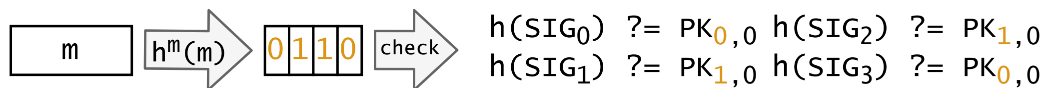


Figure 1.1 An overview of key generation, signing and signature verification algorithms. For simplicity, the variant with parameter $n = 4$ is presented, meaning that the output size of the hash functions is four bits. The signer hashes the message and, based on the bit value at the given input, selects the secret key from the first set or the second; this forms the Lamport signature. The verifier does a similar thing — the value of the bit determines if the hashed value from the signature is checked against the first or the second set of public keys.

Signing To sign the provided message m, hash the message using the h^m function to obtain a digest $d = h^m(m) = (d_0, ..., d_n)$. Now for each bit of d, assemble the signature σ based on the d_i value as follows $\sigma = (\mathrm{SK}_{d_0 n+0}, \mathrm{SK}_{d_1 n+1}, ..., \mathrm{SK}_{d_n n+n})$.

Verifying To verify the message m with the signature $\sigma = (\sigma_0, ..., \sigma_n)$ and the public key $\mathrm{PK} = (h^k(\mathrm{SK}_0), .., h^k(\mathrm{SK}_{2n}))$, first hash the message using the h^m function to obtain a digest $d = h^m(m) = (d_0, ..., d_n)$. Now for each $i \in (0, ..., n)$ compute hash value of $h^k(\sigma_i) = \mathrm{PK'}_i$ and compare it to $\mathrm{PK}_{d_i n+i}$. If all these values match, the signature is valid.

Security The security of the Lamport scheme is based on the security of the one-way and hash functions used. Provided that cryptographic hash function h is used for both message $h^m : \{0, 1\}^* \to \{0, 1\}^n$ and key function $h^k : \{0, 1\}^n \to \{0, 1\}^n$, the security against generic attacks is at least the minimum of securities for the three mentioned properties. Those are collision resistance, pre-image resistance and second pre-image resistance. Since collision resistance implies second pre-image

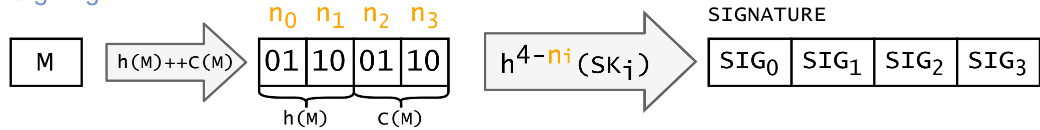
resistance, it is sufficient to consider only collision and pre-image resistance. Importantly, the single key pair can sign only a single message because a single signature reveals a significant part of the secret key, and the security decreases dramatically. One signature reveals the whole half of the secret key, and the probability of an adversary forging a valid signature for some message is high already.

Winternitz one-time signature scheme

The Winternitz scheme idea was first published by Merkle [24] in 1989. However, the scheme is not thoroughly described in that publication; in full detail is described in [25]. Compared to the Lamport one, this scheme can save a lot of signature size. It can be 4, 8 or, in extreme cases, 16 times smaller due to the tradeoff between computation time and signature (and also key) size. The main idea here is that the message with the appended checksum is fragmented into pieces of specific bit size, and these bits, as unsigned values, determine how many times the signer and verifier chain hash the values to get the value from the public key. The checksum here is important because it stops the attacker from forging a valid signature after seeing the original signature. In this section, repeated hashing is denoted by the upper index number in parentheses — the input of i-th iteration is the input for the $i+1$-th. This procedure is nothing else than a mathematical function composition. As an example, consider notation $h^{k(3)}(x) = h^k(h^k(h^k(x)))$; this means that we chain-applied the function h^k three times.

Primitives The scheme uses a cryptographically safe random number generator for secret key generation and a one-way function $h^k : \{0,1\}^n \to \{0,1\}^n$ for public key derivation and verification. Also, it uses a cryptographic hash function $h^m : \{0,1\}^* \to \{0,1\}^n$ for generating a digest of a message m. Again, the real-world implementations conveniently use one function h at both roles.

Parameters The scheme uses the security parameter n that determines the size of secret key numbers and the output size of the hash function h^m. The $w \leq 2$ parameter sets the tradeoff between computation time and signature size. It determines how many times the secret keys are hashed initially and how many bits it signs with one secret key number. For the given w, one key number signs $log_2 w$ bits. Based on the bit size of the message m, we compute how many segments the signature will contain both for the message itself (l_1) and for the checksum (l_2):



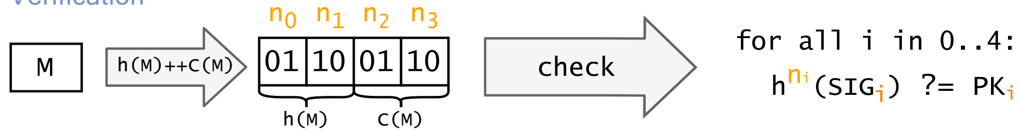


Figure 1.2 The Winternitz signature scheme overview with parameters $w = 4$ and $n = 4$. With that said, a hash function has an output length of four bits and keys are made of two random numbers; it corresponds to parameter $w = 4$ and therefore $log_2 4 = 2$. Thus, the signer divides the message digest with the checksum into two-bit blocks representing an unsigned integer that determines how many times the hash function should be applied to the signature pieces; this chain-hashed secret key forms the signature. The verifier also hashes the message and splits the digest just as the signer; after that, the verifier completes the intermediate chain-hashed secret keys to the secret keys that were hashed four times. These w-times hashed values are checked against the public key for each segment.

$$k_1 = \left\lceil \frac{m}{\log_2 w} \right\rceil k_2 = \left\lfloor \frac{\log_2(k_1(w-1))}{\log_2 w} \right\rfloor + 1$$

Key generation Using the PRNG, generate $k = k_1 + k_2$ random numbers of size n; this is the secret key $\mathrm{SK} = (\mathrm{SK}_0, ..., \mathrm{SK}_k)$. Compute the public key by hashing the secret key numbers with h^k w-times; i.e. $\mathrm{PK} = (h^{k(w)}(\mathrm{SK}_0), .., h^{k(w)}(\mathrm{SK}_k))$.

Signing To sign the provided message m, hash the message using the h^m function to obtain a digest $d = h^m(m) = (d_0, ..., d_{k_1})$. Compute the checksum of the digest $C = (c_0, ..., c_{k_2})$ as

$$C(d) = \sum_{i=0}^{k_1-1} (w - 1 - d_i)$$

and also split it into $l = log_2 w$ bit segments. The concatenation of the digest and the checksum $d' = (d_0, ..., d_{k_1}, c_0, ..., c_{k_2})$ is now split into k unsigned integers of size l bits, i.e. $d' = (v_0, ..., v_k)$. Then, the signature σ is produced based on the values v_i as $w - v_i$-times chain hashed values. The signature looks like this — $\sigma = (h^{k^{(w-v_0)}}(\mathrm{SK}_0), ..., h^{k^{(w-v_k)}}(\mathrm{SK}_k))$.

The checksum guarantees that the attacker cannot forge a valid signature after seeing the original signature. Without the checksum, the attacker could sign any message with a smaller value of all segments. The checksum guarantees that if signing the message with all values less than the message in the original signature, the checksum will contain at least one segment with greater value and thus, the attacker cannot compute its chain-hashed value for the forged signature because only the higher chain-hashed value was revealed.

Verifying To verify the message m with the signature $\sigma = (\sigma_0, ..., \sigma_k)$ and the public key $\mathrm{PK} = (\mathrm{PK}_0, ..., \mathrm{PK}_k)$, first hash the message using the h^m function to obtain a digest $d = h^m(m) = (d_0, ..., d_n)$. Now split the digest into k unsigned integers of size l bits, i.e. $d = (v_0, ..., v_k)$. Then for each signature part, the value from the public key is produced based on the values v_i as v_i-times chain hashed values σ_i from the signature. The signature is valid if the chain-hashed values match the i-th element of the public key. In other words $\forall i \in (0, ..., k) : h^{k^{(v_i)}}(\sigma_i) = \mathrm{PK}_i$.

Security The scheme's security is again derived from the security of the used one-way and hash functions. Just as in the case of the Lamport scheme, it is the minimum of securities from collision and pre-image resistance. Also, the key pair can be used only to sign a single message because only one signature reveals a significant part of the secret key (its partly chain-hashed values), and the security decreases dramatically. The probability of an adversary forging a valid signature for some message is high.

1.3.2 Few-time signature schemes

One-time signature schemes were already introduced; their main disadvantage is that they can securely sign just one message. It was also mentioned that one-time schemes could be used to construct a few-time signature scheme. This section introduces one such approach — Merkle Signature Scheme (MSS) [26]; an extended variant of this scheme (XMSS) [27, 28] is now a recommended option by NIST [29] in the category of stateful post-quantum cryptography. After that, a stateless few-time scheme used as a core for the proof-of-concept of this thesis implementation is explained — Hash to obtain Random Subset with Trees (HORST) scheme [30].

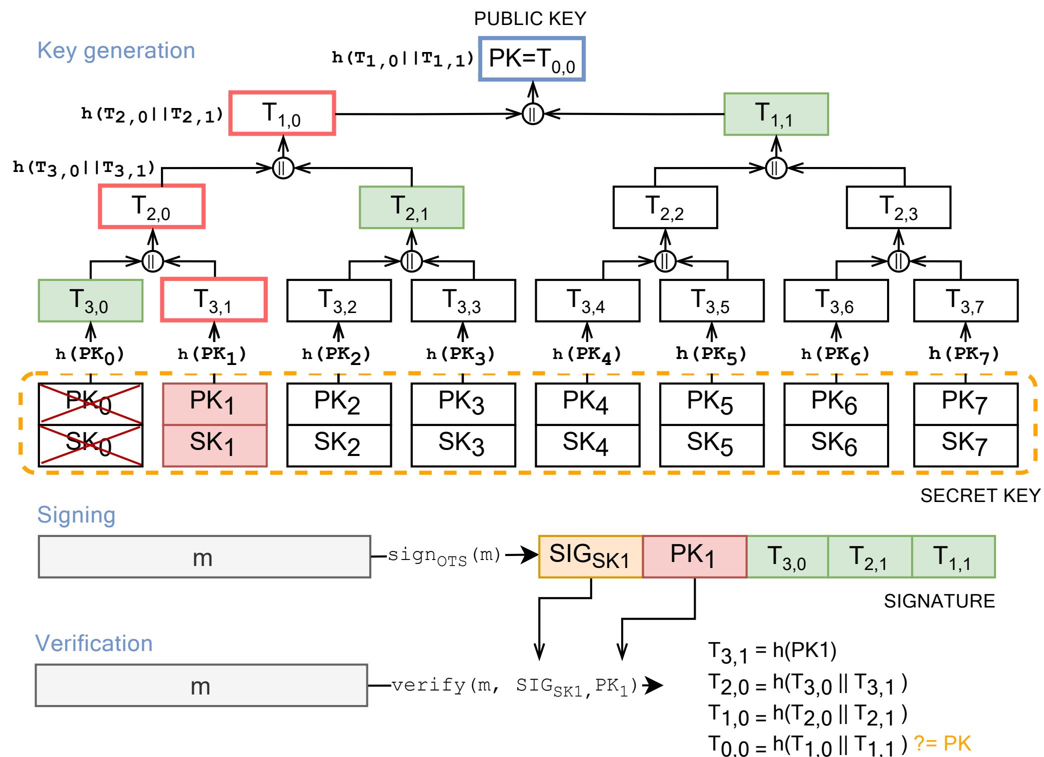


Figure 1.3 A demonstration of Merkle signature scheme key generation, signature generation and verification; the used parameters are $h = 3, l = 2^3$. The signer generates the signature with the key index 1 because the first index has already been used. The signature consists of the signature from the underlying one-time scheme used extended with the corresponding public key and necessary Merkle tree nodes for reconstructing the whole authentication path to the root. The reconstructed value is then compared to the public key.

This scheme was introduced as part of the construction of the SPHINCS signature scheme. An improved version of the HORST scheme — FORS [23] — is used in the SPHINCS+ scheme, a recommended option in the category of stateless quantum-resistant schemes.

Merkle Signature Scheme

The Merkle signature scheme was first introduced in a dissertation thesis of Ralph Merkle in 1979 [26]. The scheme offers small public keys and an adjustable number of signatures per key. This signature scheme can issue only a limited number of signs per single public key and is built upon an arbitrary post-quantum one-time scheme — e.g. Lamport signature or Winternits scheme from the previous

sub-section — with at least desired target bit security. The key idea here is that the number of expected signatures per key is selected as a power of two — for example, $l = 2^h$. Then l secret keys of the chosen one-time signature scheme are generated; the secret keys are hashed using a hash function $h : \{0,1\}* \rightarrow \{0,1\}^n$ into a basis for a public key. Then complete hash tree is built upon those hashed secret keys, and the root is published as a public key. Then for each message, one secret key is used exactly once; the signature is composed of a signature of a one-time scheme and the authentication path to the root. The authentication path is the neighbourhood of a path from the signing leaf to the root—the necessary minimum is to compute the valid root hash. Because the signature scheme must know what underlying keys have already been used, it belongs to a stateful signature scheme category. The overview of the scheme principles is shown in the figure 1.3.

Primitives The scheme needs a cryptographic hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ for hashing a Merkle tree and OTS private keys. Additionally, it needs the primitives required by the underlying one-time scheme.

Parameters The scheme uses the security parameter n that determines the output size of the hash function h and a number of leaf nodes $l = 2^h$. Also, it requires parameters for the underlying one-time signature scheme.

Key generation Generate l keypairs using the underlying scheme, i.e. $(\mathrm{SK}_i, \mathrm{PK}_i) \forall i \in (0, \dots, l-1)$. For each OTS public key, compute $T_{h,i} = h(\mathrm{PK}_i)$; these are the leaf nodes in the Merkle tree. The tree root serves as the public key. The rest of the tree is computed recursively as follows $T_{j,i} = h(T_{j-1,2i} +\!\!+ T_{j-1,2i+1})$. To be consistent with our pseudo-codes that follow, the $+\!\!+$ sign denotes string concatenation as an alternative to the usual $||$ sign. The root of the tree is then exposed as the public key.

Signing The signing process starts with the OTS scheme signing the message m yielding signature σ_{OTS}. The next part of the signature is OTS public key that can be used to verify σ_{OTS} for the message m. Additionally, the authentication path and the key index must be included. The authentication path is the sequence of neighbouring nodes on the path from the key leaf node to the root; the minimum number of values to correctly hash from the bottom to the top. In the figure 1.3, it can be seen what the authentication path for the key index 1 looks like — $T_{3,0}, T_{2,1}, T_{1,1}$. The final signature for i-th key looks like this $\sigma = (\sigma_{OTS}, \mathrm{PK}_i, T_{auth_0}, \dots, T_{auth_{d-1}}, i)$.

Verifying The verifier checks the validity of the OTS signature for the message m with the attached OTS public key PK_i. If it matches, the process can continue — the OTS public key hash is computed as a leaf node $T_{h-1} = h(\mathrm{PK}_i)$. Then the path to the root is computed using the attached authentication nodes. To determine if the authentication node is to be concatenated from left or right, the key index i is used; the index in the given layer of the tree determines this — even indices are those concatenated from the left side. Once the root value is computed, it is compared with the public key; if it matches, the whole signature is valid.

Security The security of the Merkle signature scheme lies, of course, in the underlying signature scheme. Besides that, the scheme requires a pre-image and second pre-image resistance of function h. There is no way to exploit collisions in the Merkle tree. Assuming the function h is ideal, the security of the Merkle scheme itself is 2^n.

An overview of the situation for parameters $h = 3, l = 2^3$ is captured in the figure 1.3. The process of signing and verifying with key index 1 is presented.

HORST scheme

The Hash to Obtain Random Subset with Trees scheme is the next in the family of few-time signatures. This scheme is used as a signature scheme for the proof-of-concept implementation in the protocol. It was selected for its reasonable tradeoff between complexity, practical usability and security. Also, it offers small public keys that need to be distributed often in the proposed protocol.

HORST was used as one of the building blocks for the original SPHINCS signature scheme in 2015 [30]; it is just a minor extension of the HORS scheme [31]. The authors of SPHINCS used a Merkle tree to reduce the size of the public key to just one hash; on the other hand, this increased the size of signatures a bit. The main idea behind this scheme is that there is a massive amount of k-element subsets in $t = 2^\tau$ set, provided that the parameters are well selected.

In the HORST scheme, the private key consists of many secret key elements, and the public key is derived from the corresponding hash values using a Merkle tree. When signing a message, a small random subset of the secret key elements is chosen based on the message hash, and these elements are revealed as part of the signature. The signature also includes authentication paths for each revealed secret key element, allowing the signature to be verified using the public key. The terms Merkle tree and authentication paths are the same as in the Merkle signature scheme; they were described in the section 1.3.2. The overview of scheme principles is visualised in the figure 1.4. One possible format of signature, the format that is used in the PoC implementation, is shown in the figure 1.5.

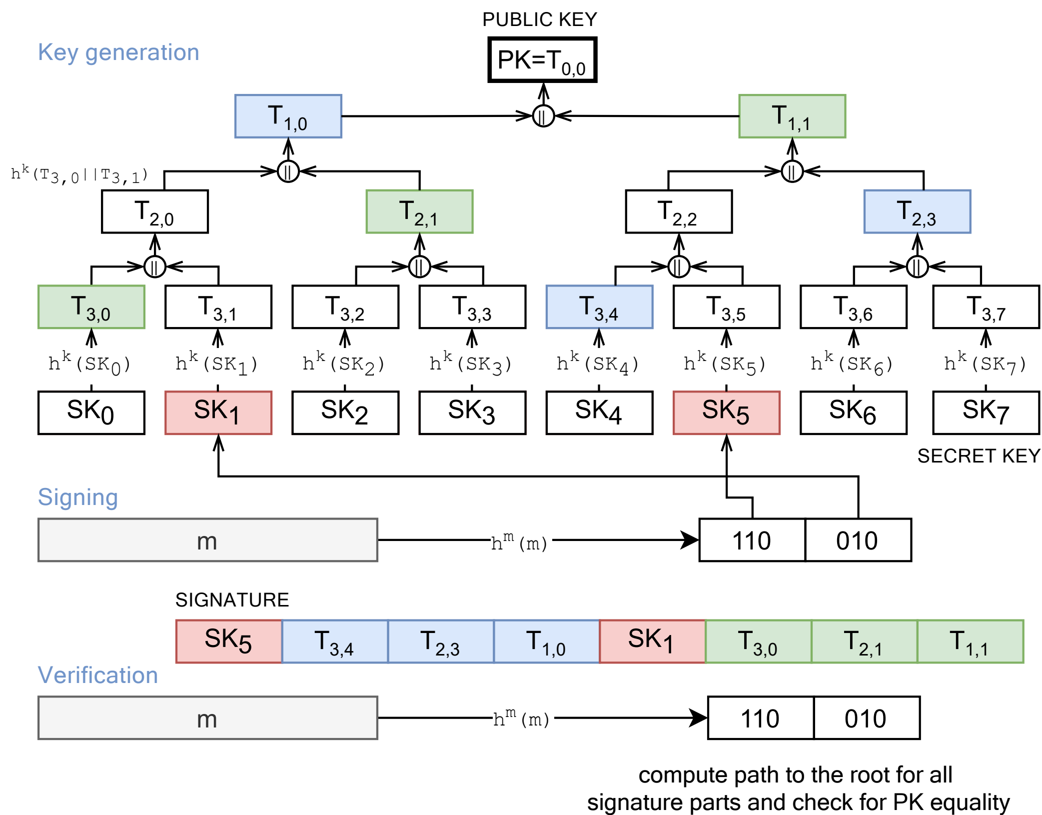


Figure 1.4 An overview of HORST key generation, signing and verifying displayed for the parameters $k = 2, \tau = 3$. The signer reveals a few secret keys based on parts of the message and delivers the necessary Merkle tree nodes so that the verifier may retrace the whole hashing chain to reconstruct and verify the public key.

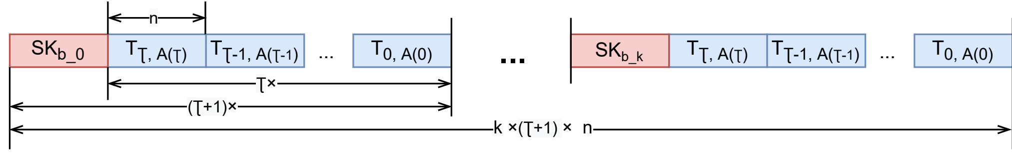


Figure 1.5 An example of HORST signature format; it uses k elements of the 2^τ set to form a signature. Each element is followed by τ hashes of size n bits — i.e. the size of the hash function output used to build the Merkle tree. These elements form an authentication path and are necessary to calculate the root node that should be equal to the public key.

Primitives The scheme uses a cryptographically safe random number generator for secret key generation and a cryptographic hash function $h^m : \{0,1\}^* \rightarrow \{0,1\}^n$ for hashing messages and $h^t : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ for hashing concatenations while building a Merkle tree over the hashed secret keys.

Parameters The scheme's parameters are n, determining the size of hash function output, τ specifying the depth of the Merkle tree and k, which tells how many secret key pieces are revealed with each signature.

Key generation To generate a keypair, $t = 2^\tau$ random numbers of size n must be generated with the CS PRNG; this is the secret key $\mathrm{SK} = (\mathrm{SK}_0, \dots, \mathrm{SK}_t)$. Then every secret key is hashed with the h^t hash function, and these values $(T_{\tau,0}, \dots T_{\tau,t})$ are used as leaf nodes of the Merkle tree. The next step is to compute the whole Merkle tree; it is the same process as in the Merkle signature scheme — concatenation of child nodes is hashed with the h^t function and stored as the parent value, i.e. $T_{i,j} = h^t(T_{i-1,j*2} \mathbin{+\!\!+} T_{i-1,j*2+1})$.

Signing To sign a message, hash it $h^m(m)$ and divide the output into τ-bit blocks $h^m(m) = (b_0, \dots, b_k)$. Now represent each b_i as an index to $t = 2^\tau$ elements and form k-sized subset based on the indices b_i. For each of the k elements, add the secret key SK_{b_i} into the signature, followed by the authentication path for the path from this secret key to the Merkle tree root. The final signature $\sigma = (\mathrm{SK}_{b_0}, T_{\tau,A(b_0,\tau)}, \dots, T_{0,A(b_0,0)}, \mathrm{SK}_{b_1}, T_{\tau,A(b_1,\tau)}, \dots, T_{0,A(b_1,0)}, \dots)$ looks like this — figure 1.5. The $A(b,i)$ denotes the corresponding index of the authentication node for the secret key at index b on the given tree layer.

Verifying First, compute the hash of a message $h^m(m)$. To verify the signature σ, divide the output into τ-bit blocks as follows $h^m(m) = (b_0, \dots, b_k)$. Then represent each b_i as an index to $t = 2^\tau$ elements; use this index to determine what

authentication path to the root is one reconstructing. Use the h^k function on the secret key and then use the authentication parts to compute the root hash and compare it with the public key. If the key matches, the signature is valid.

Security The security of the HORST scheme lies in the hash functions used, specifically in their pre-image and second pre-image resistance. The size of hash function output sizes again determines these resistances. Besides that, this scheme can also be broken by simply being able to sign the given message by forming a subset from already published secret key pieces. Naturally, the probability that an adversary can forge a valid signature grows with every signed message; r denotes the number of already signed messages by the current key. That implies that the security decreases with every signature. As analysed in this publication [32], the HORST scheme offers a decent bit of security against brute-force attack —

$$k(\log_2 t - \log_2 k - \log_2 r).$$

However, it is vulnerable against adaptive chosen-message attacks where the security decreases dramatically; bit security against the adaptive attack can be approximated as

$$\frac{q^{r+1}}{r!}(\frac{kr}{t})^k.$$

This vulnerability can be eliminated with modifications proposed as FORS scheme [23]. Additionally, the scheme is vulnerable to so-called 'weak messages' attacks; specifically targeted messages that employ only a reduced number of elements forming the subset. As described in the next paragraph, these can be eliminated by scheme modification.

There are also many extensions proposed to HORS and HORST. Be it PORS [33], the scheme that eliminates the vulnerability against weak messages where the subset is not derived from message hash but rather from a pseudo-random generator; the seed is provided as a part of a public key. The other extensions are HORSIC [34] or HORSIC+ [34], that has been proposed in 2021 as an extension to HORSIC. Also, the Forest of Random Subsets (FORS) [23] scheme was introduced as a substitution to the HORST scheme for SPHINCS$^+$. It uses a forest of Merkle trees instead of just one to eliminate vulnerability against adaptive chosen-message attacks.

Although this scheme may seem vulnerable, it is used as a basis for proof-of-concept implementation for this thesis. The rationale is this — the underlying scheme can be replaced with any few-time signature; HORST was chosen as one with decent security and implementation simplicity ratio. Moreover, the adaptive or weak message attacks are irrelevant in this setup because no messages can be signed for the potential attacker. Finally, it is just proof-of-concept of a thesis

where the focus is to verify if few-time signatures based on hash functions are usable for the problem of authenticated streaming and broadcasting.

1.3.3 Current development in hash-based signatures

To provide a sneak peek into state-of-the-art candidates aspiring to become post-quantum alternatives to currently used digital signatures, we discuss some of the recommendations of the National Institute of Standards and Technology (NIST) experts. NIST has been working on standardising post-quantum cryptography through its Post-Quantum Cryptography Standardization project since 2016. Currently, SPHINCS+ is selected as a candidate for standardisation in the category of hash-based and stateless signature schemes [2]. As for candidates for stateful schemes, NIST recommended two schemes: eXtended Merkle Signature Scheme (XMSS) and Leighton-Micali Signatures (LSM) [29]. These schemes have demonstrated robust security properties and practical performance characteristics, making them strong candidates for protecting digital communications against potential quantum attacks.

SPHINCS and SPHINCS$^+$

At its core, the SPHINCS [30] scheme relies on a virtual hypertree organised into layers where each layer contains W-OTS+ trees and where each leaf signs the root of the underlying tree. In the last layer (layer 0), HORST (FORS in SPHINCS$^+$) trees are used for the final signature. The issued signature then determines one path through the hypertree, allowing the verifier to recreate it and check the match with the public key.

SPHINCS+ [23] is an improved version of SPHINCS that addresses some of the performance and security limitations of the original scheme. It introduces several optimisations, such as using FORS instead of HORST and multi-target attack protection. These improvements result in smaller signatures and keys and faster signing and verification times. SPHINCS+ also offers various parameter sets to balance security and performance using different use cases.

The scheme can sign a very high number of messages with one key. Generally, it is 2^h, where h is the total height of a hypertree. The recommended configurations imply that at least 2^{64} messages can be signed securely. The authors proposed various scheme instantiations for multiple security levels, as seen in the NIST submission paper [3]. Although the security properties are very appealing, the drawback is the signature size and computational complexity of signature

[2] https://csrc.nist.gov/projects/pqc-dig-sig
[3] https://sphincs.org/data/sphincs+-specification.pdf

generation. This makes this scheme bit unpractical for the protocol proposed in this thesis, which prefers more lightweight signature schemes.

eXtended Merkle Signature Scheme

XMSS [27] is an extended version of the previously described MSS scheme [26]. The underlying one-time signature scheme used is W-OTS+ [35]. The construction is similar to a normal Merkle scheme with a few differences. When computing the parent node, one first applies a XOR function with randomly generated masks on the values before concatenating and hashing; this is a so-called XMSS tree. This masking leads to the fact that the hash function used on trees does not have to be collision resistant. Also, leaves of the XMSS tree are not hashes of a one-time signature public key but roots of another not necessarily full binary tree called L-tree. This modified XMSS tree compresses the OTS public key to a single hash value; it may be incomplete from the right because the key may not be sized in powers of two.

It can sign 2^h messages, where h is the depth of the main Merkle tree; and the scheme's parameter. A second pre-image resistance of the underlying hash function determines the security of the whole scheme. The disadvantage of this scheme is that it is stateful, and the next index of a key to use must be part of the secret key.

Chapter 2

Authenticated streaming with hash-based signatures

With knowledge of the necessary context and cryptographic primitives used as building blocks, this chapter states the purpose and scope along with requirements for the proposed protocol and the required protocol interface. The detailed specification of the protocol follows — firstly, network layer formats are presented then all three roles are specified for the required interface. Also, important properties and parameters are discussed. The chapter concludes with a discussion about possible trade-offs accessible through protocol parametrisation.

2.1 Design Criteria

Before delving into the protocol design specification, it must be known what the primary purpose of the protocol is, what is and what is not the responsibility of the protocol (scope), what the protocol shall and shall not do (requirements), and how the users can interact with it (interface). All these aspects are presented in the following section.

Summary of the main constraints The main goal is to transport a stream of data from a sender to a receiver and allow the receiver to verify the authenticity of the data while assuring that an adversary did not mangle the data stream.

First, since we aim for sender offloading, we must avoid any connection-oriented communication. Tracking of connections would complicate the sender logic and place unnecessary expectations on the sender-receiver communication that might make retranslation unnecessarily complex.

Second, since we aim for complete decentralisation, we avoid using any central or globally available key store; this leaves us with a petname-style identity

system [36] that uses the public keys as identities. Moreover, this is complicated by the ephemeral nature of the few-time signature keys, which forces us to update public keys and, thus, also identities continually. An important part of the protocol allows re-authentification of the senders even after substantial updates of their identities.

Finally, the hash-based signatures present a possibly significant bandwidth overhead; we therefore have to tune the size of the signed chunks to trade off the bandwidth overhead, authentication latency, ephemeral key consumption rate, and several other variables.

2.1.1 Purpose

The proposed protocol shall be used to check the integrity and authenticate data streams from the sender to receivers that have actively subscribed to receive the data. The sender shall be able to generate and persistently keep his sender identity that receivers will trust and accept. The protocol shall allow the sender to stream the data to multiple actively subscribed receivers.

Receivers shall be able to subscribe to the specified data source. After accepting and *trusting the first message* from that source, they shall have subsequent incoming messages authenticated if they were originally sent from the same identity. The number of missed messages from the given identity shall limit this guarantee of re-authentication; this should be configurable.

Additionally, the protocol shall support a distributor role; such is meant to distribute messages sent by a sender who is already known-identity to the distributor and forward them to receivers that subscribed to that data via the distributor.

The overall cooperation of the roles is sketched in the figure 2.1. The arrows show the flow of a message from the original sender to all active receivers.

2.1.2 Scope

The proposed protocol shall serve as a layer between a user application that leverages it to implement secure data streaming and the abstract building blocks this protocol uses. These building blocks are the underlying few-time signature scheme and a network interface for sending packets. It is designed to be implemented as a user-space application without special permissions.

The underlying signature scheme is not in the scope of the protocol. The protocol shall operate with an arbitrary selection of the scheme if it is a post-quantum few-time signature scheme. The network interface is not in the scope of this protocol either.

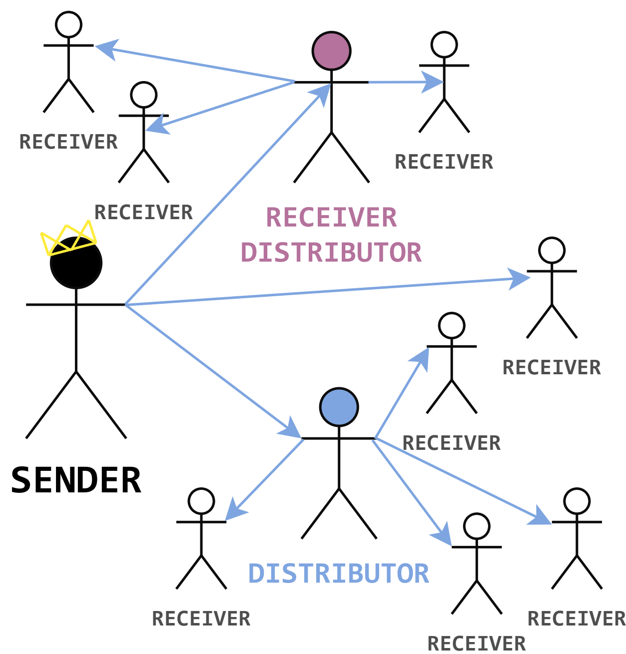


Figure 2.1 An example flow of pieces from the sender to all receivers while using the protocol. One original sender is streaming pieces containing the original data stream. There is the receiver who is also a distributor; this node is re-distributing the authenticated pieces from the known sender to other receivers that asked this node to do so. Additionally, there is a node that is just a distributor — the node receives, authenticates and re-distributes the pieces but does not deliver them. This re-distribution reduces the load on the sender and, moreover, can reduce the network bandwidth compared to unicasting the data to all receivers from the sender.

2.1.3 Main abstractions

The construction is based on an abstraction of a signature scheme and a network interface. The following sections specify the required properties and interfaces.

- **Post-quantum few-time signature scheme**

  The scheme to be used in the protocol shall be quantum-resistant and few-time. The scheme shall, for the given parameters — including minimum required bit security— yield a maximum number of signatures one key can generate.

  As for the interface to interact with the scheme, three methods are required — `generate_keypair`, `sign` and `verify`.

- **Network interface**

  Since the main point of the proposed protocol is data streaming over computer networks, it requires a network interface. The interface must support sending packets between nodes with addresses. Reliability or confidentiality is not required. An example of such a protocol can be UDP as described in RFC 768 [37].

Signature scheme interface

`generate_keypair` The `generate_keypair` procedure shall return a new keypair for subsequent signature generation and verification.

- Input: None.
- Output: A new keypair that has two parts — secret and public. The secret key shall be usable as an input to `sign` and the public key as an input to `verify`.

`sign` The `sign` procedure shall generate a valid signature for provided data with a provided secret key.

- Input: An array of bytes to sign and a secret key to sign it with.
- Output: The provided array of bytes and a valid signature generated with the provided secret key.

`verify` The `verify` procedure shall verify the validity of a signature for provided data.

- Input: A signed array of bytes, a signature and a public key.
- Output: The provided array of bytes and a boolean flag indicating if the signature is valid for the provided bytes.

2.1.4 Requirements

In this part, the guarantees provided by and requirements for the protocol are stated; these set bounds for the subsequent protocol specification. The requirements are as general as possible without unnecessary technical details irrelevant to the functionality of the protocol. First, the general requirements are declared, followed by requirements relevant to the specific roles.

Protocol guarantees and invariants

The protocol manages identities generated by users, which can uniquely cryptographically sign and authenticate network packets using a fast quantum-resistant cryptography algorithm based on few-time hash signatures, suitable for use in high-performance and low-latency network streaming.

The authentication must not rely on any other data transfer than the transported packets (i.e. there must not be a central authority to distribute the identities, etc.). The protocol must accordingly include the authentication information in the signed packets, and anyone who receives a packet must be able to uniquely identify the sender, allowing authentication of the same sender in some subsequent packets received from the sender. By receiving and decoding a sufficient amount of messages over time, the receiver must be able to authenticate packets in the subsequent possibly infinite message stream. For convenience, the protocol will additionally have to be able to subdivide the messages into fragments of sizes suitable for being transferred over the network. The protocol does not guarantee the confidentiality of the messages nor the reliability of the message transfer. Confidentiality is typically not required for broadcasting applications and may be optionally implemented in a higher layer, and transfer reliability may be implemented by utilising a stronger fragment delivery protocol.

Sender implementation

Sender software should provide identity management functionality, mainly the generation of the few-time signature keys and updates of the key store. All operations must be implemented to prevent losing the identity information, e.g., due to hardware faults and other events.

Receiver implementation

Similarly, as with the sender, the receiver software should prevent losing the sender-authenticating information due to errors. For user convenience, it must keep a database of the identities and map the received identities and packets to a persistent identity useful for the software and authentication status of the

message. The receiver implementation must expect packet loss and reception of spoofed or malicious fragments. Each message to be delivered should be tagged as one of the following:

- *Authenticated by identity X* — the identity X signed the message..
- *Certified by identity X* — the identity X certified the key that signs this message..
- *Unauthenticated* — the message is not proven to be signed by nor certified by the target identity.

Authenticated and certified messages should be delivered with strictly increasing sequence numbers, but the delivery should not be blocked too long if some sequence number is missing; the delivery should continue with subsequent messages after some reasonable time elapses.

The requirements on a distributor role

Receivers also should be capable of distributing unaltered packets from receivers subscribed to them. The distributor shall work as a transparent proxy between the original sender and the receiver. Receiving data from a distributor should be as secure as receiving it directly from the original source.

2.1.5 The interface of the protocol

For a protocol user, it shall be as simple as possible to use it in the applications. The protocol instance must allow initial parametrization; the underlying signature scheme also determines a subset of the parameters. Then a way to send the next message from the data stream must be available. On the other end — at the receiver, there must be a way to have authenticated messages delivered.

`new_sender` Constructs and configures the protocol instance of a sender with the provided parameters.

- INPUT: Protocol and signature scheme parameters relevant to the sender role.
- OUTPUT: An instance that can broadcast the data to receivers.

`new_receiver` Constructs and configures the protocol instance of a receiver with the provided parameters.

- INPUT: Protocol and signature scheme parameters.
- OUTPUT: An instance that can receive and/or distribute authenticated messages.

`broadcast` Securely broadcast the provided data to all subscribed receivers.

- Input: An array of bytes as data to be broadcasted.
- Output: None.

`receive` Waits for a message from the subscribed receiver, does the authentication and integrity check, and returns the message with an appropriate authentication flag and the sender's identity.

- Input: None.
- Output: It returns an array of bytes as the message, the status of verification — verified by identity X, certified by identity X or unauthenticated — and a sequence number of this message.

2.2 Specification of the protocol

This section provides a detailed specification for the proposed protocol. The protocol fulfils the requirements in the scope described in the previous subchapter section 2.1 to achieve the stated purpose. The interface is designed to match the interface specification. The section starts with specifying different networking layers and their data format. Then a thorough specification of each protocol role is presented. Towards the end of the section, the reader finds a discussion on the vital protocol property — time to re-authenticate the stream for an already-known identity, given that a specific number of messages were missed. The very end of this section summarises the protocol parameters.

Briefly, the protocol works as follows: The sender implementation splits the *data* stream into messages; these are encapsulated into *pieces* that can be delivered as separate units to receivers. All pieces are cryptographically signed so that their authenticity and integrity may be verified. The key for signing the current piece is sampled from the layer hierarchy. Additionally, pieces contain protocol metadata, such as announcements of other public keys that the receivers use to update their key database and to update known identities. To fit the pieces into the typical network packets, the sender splits them into *fragments*, which the receiver has to reassemble before verification. Fragments are transmitted in *datagrams* using the underlying network interface.

The critical aspect of the protocol lies in the appropriate layer selection to sign the next piece. The keys on each layer have a limited number of signatures they can generate — this results from using few-time signatures —, and then the key must be replaced by a new one. To guarantee a potentially infinite number of signatures for the data stream, the sender keeps a hierarchy of key layers where the probability of signing with the given layer decreases when getting closer to

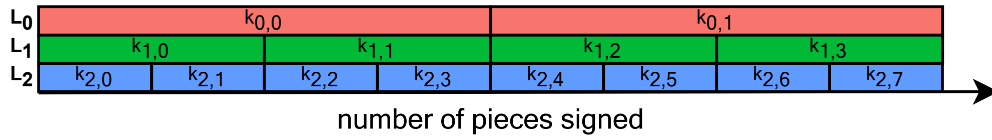


Figure 2.2 The expected key lifetime ratios on different layers with `KEY_RATIOS = [(4,0), (2,0), (1,0)]`. One can see that keys from L_0 sign two times less frequently than L_1 and four times less frequently than layer L_2.

the first layer; this implies that the keys in layers with lower indices live longer than those with higher indices since they sign less frequently and their number of secure signatures is not exhausted that fast. The `KEY_RATIOS` parameter determines the ratio of the expected key lifetimes for each layer. The example hierarchy is visualised in the figure 2.2; there, an exponential ratios `KEY_RATIOS= [(4,0), (2,0), (1,0)]` are used. The first integer in the tuples in the list determines the lifetime ratio — $4:2:1$ in this case. Consequently, keys from the L_0 layer should be active two times longer than those from L_1 and four times longer than L_2 keys or said in terms of probabilities — the probability of signing with a key from layer L_2 is two times lower than that of using L_1 and four times lower than the probability of signing with L_0 key.

This layer hierarchy, combined with the continuous announcements of the keys that will be used once the active ones are exhausted — i.e. no more secure signatures can be generated, guarantees that the sender can sign a potentially infinite number of pieces with keeping his initial identity. The identity is the whole chain of used keys that have certified each other since the identity was generated. The important feature is that the stream of pieces certifies itself. Since the initial trust in the first piece the receiver accepted, the subsequent pieces can be verified by the keys that the receiver already knows due to announcing the keys that will be used in the future. Thanks to this, receivers can keep the database of active keys as well as keys that were announced and will be signing pieces in the future. This way, the receiver keeps the chain of identity keys alive even with significant piece losses — these will certainly happen due to the receiver going offline and reconnecting later. The number of pieces lost that the receiver withstands without losing the sender's identity can be configured. The receivers do not have to remember the whole chain but only the most recent segments while transitively spreading the existing sender identity. These two crucial mechanisms are discussed in detail in the section 2.2.2 and section 2.2.3, respectively.

The general notation used in this section This section uses a lot of pseudo-code to specify the protocol's behaviour. For brevity, the functions use a few one-word variables; `s` for a sender and `r` for a receiver. Also, some variables are prefixed with `SH.`; the `SH` denotes a set of shared structures that any pseudo-code can access. The reader also notices the variable named `p`, which stands for protocol parameters; the parameters are introduced as needed in the specification and then summarised at the end of this section. Also, be wary that due to the protocol generality, the parameters of the underlying scheme are all compressed inside the parameter `FTS`; this is because every scheme uses different parameters. For example, the `get_keys` function takes the `FTS` parameter as an argument to generate keys because the underlying scheme determines the form of the keys. The specific implementation for every scheme differs; therefore, these details related to the scheme used are considered implementation details.

2.2.1 Network formats

The protocol's responsibility starts whenever the user application calls the `broadcast` function. The data transmission works in layers — at the sender side, starting from a data stream to messages, from messages to pieces, from pieces to fragments and finally, from fragments to datagrams. The receiver uses a reversed order of layers. The hierarchy of the data encapsulation is visualised in the figure 2.3.

Pieces and fragments The most important part is the piece; it has a header and a message. If the data above is small enough, it can fit into a single piece; this depends on the `MAX_PIECE_SIZE` parameter, and the actual piece header size depends on other protocol parameters (most notably, the security of the underlying signature scheme). The format of a piece can be seen in the figure 2.4. A piece is an essential part because it represents one signed message in the context of the protocol. One unique signature must be issued for each piece to depart to receivers. It contains, among other parts, the message itself, a signature and a sequence number. Moreover, it holds public keys that serve receivers as a way to reauthenticate in the future if they miss some pieces. The number of keys attached to a piece is in the header too. Each public key is prefixed with an unsigned 8-bit integer, denoting the layer it belongs to — as shown in figure 2.4. Layers are numbered from 0 to n, and 0 is always the longest-living layer. The piece is then split into a minimum number of *fragments*.

A fragment splits pieces into smaller blocks, adds the header, and uses it as a payload to a set of *datagrams*. A fragment format can be seen in the figure 2.5. A datagram is a term for a block of data that is of suitable size for transfer via the underlying network interface.

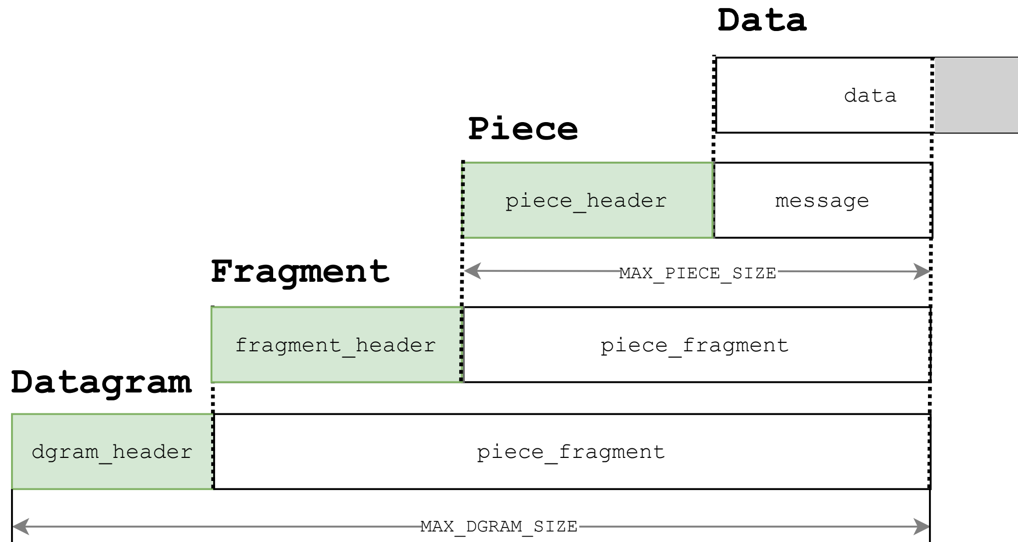


Figure 2.3 Data encapsulation is shown from top to bottom. The datagram refers to a block of data that is suitable for transfer via the underlying network interface. The formats of pieces and fragments are displayed in figure 2.4 and figure 2.5, respectively.

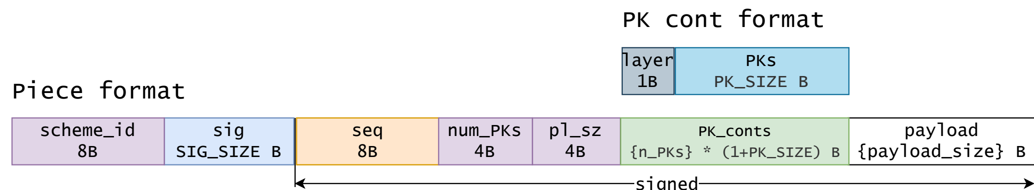


Figure 2.4 The diagram shows the format of a piece that is subsequently fragmented into fragments (see figure 2.5).

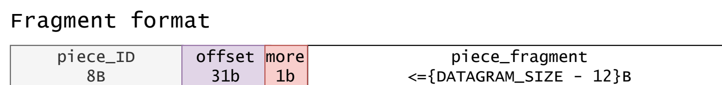


Figure 2.5 The format of a fragment that is then passed to the network interface for the transfer.

	depleted		active	future	
L_0	$K_{0,-2}$	$K_{0,-1}$	$K_{0,0}$	$K_{0,1}$	$K_{0,2}$
L_1	$K_{1,-2}$	$K_{1,-1}$	$K_{1,0}$	$K_{1,1}$	$K_{1,2}$
			...		
L_j	$K_{j,-2}$	$K_{j,-1}$	$K_{j,0}$	$K_{j,1}$	$K_{j,2}$

Figure 2.6 The state of key layers after the initialisation and population with the keys; the situation is shown for $PRE_CERT = 2$ and for j layers. The keys with the negative second index are never actually used but are attached to the pieces as if they were; this keeps the sizes of pieces consistent even when the stream session has just started.

2.2.2 Sender role

To initialise an identity, the sender first tries to load it from a file; if that fails, the CS PRNG is initialised with the provided seed parameter, and the sender uses the CS PRNG to generate the keys required to form the first layers of the algorithm.

To broadcast data, the sender splits the data into pieces, signs all pieces with selected keys while updating the key store with key use information, appending selected public keys for certification, and generating new keys as needed. The pieces are then split into fragments and passed down to the network stack.

The complete process is summarised in algorithm 1. This section walks through each beforementioned point and thoroughly specifies it.

Constructing a new sender

The constructor of the sender role contains some simple functions that are to be explained here without unnecessary technical details.

`load_sender` The `load_sender` function loads the previously stored state of a sender instance from the file located at the location specified in `ID_FILE`. The returned instance should resemble precisely the same state as when it was stored by calling the function `store_sender`, including the state of PRNG and key state information.

`seed_csprng` This function correctly initialises and seeds the cryptographically safe pseudo-random number generator of a type defined by the underlying signature scheme; this type is a part of the parameters for the given signature scheme denoted by the `FTS` parameter that groups all these.

Algorithm 1 The specification for the two interface functions for the sender role. The function names correspond to those used in the section 2.1.5. The variable `params` denotes a structure that groups a subset of protocol parameters; these are summarised in the 2.3. The shared structures are prefixed with `SH`. Here, the function `active_subscribers` reads the structure shared between this thread and the one processing the active receivers — details about keeping the active receiver is in the section 2.2.2.

```
function NEW_SENDER(p) → s
    s ← load_sender(p.ID_FILE)
    if s == ∅ then
        rng ← seed_csprng(p.FTS, p.SEED)
        key_layers ← init_layers(p.FTS, p.KEY_DIST)
        key_layers ← gen_keys(key_layers, p.FTS, rng, p.PRE_CERT)
        next_seq ← 0
        s ← (p, rng, key_layers, next_seq)
    end if
    spawn subscribers_task(p, p.SENDER_ADDR)            ▷ Background task.
    return s
end function
function BROADCAST(s, data) → s
    messages ← to_messages(data, s.p.MAX_PIECE_SZ)
    result ← []
    for each m in messages do
        scheme_id ← scheme_id(s)
        seq ← next_seq(s)
        key ← select_key(s)
        pks ← select_pks(s, key)
        num_pks ← length(pks)
        payload_size ← size(m)
        piece ← seq ++num_pks ++payload_size ++pks ++m
        signature ← sign(piece, key.secret)
        store_sender(s)
        piece ← scheme_id ++signature ++piece
        frags ← to_fragments(piece, s.p.MAX_DGRAM_SZ)
        subs ← active_subscribers(SH.subs)               ▷ Shared structure
        send_fragments(subs, frags)
    end for
    return s
end function
```

init_layers and gen_keys The init_layers function prepares the structure that holds the active key layers and necessary metadata. The layers hold the keys generated by the underlying scheme FTS; the layers to sign the next piece are then sampled from the distribution defined by the KEY_DIST parameter. The gen_keys method populates the prepared layers with the corresponding number of keys based on the PRE_CERT parameter. This parameter says how many keys are to be certified in advance. In other words, when signing with the *i*-th key on the given layer, how many subsequent keys from this layer are attached to the signed pieces. Based on this parameter, this many keys needs to be generated in addition to the one to be used initially. Also, the same number of keys must be generated as those already used even though these were never used; this keeps the sizes consistent even when starting the stream session. In total, each layer needs to be populated with 2PRE_CERT + 1 keys; the example state for PRE_CERT = 2 with j layers is visualised in the figure 2.6.

Before returning either the loaded or newly constructed instance, the background task is spawned — spawn_subscribers_task; its specification can be found in the section 2.2.2.

Splitting the data into messages

The function to_messages takes the whole data and splits it to the messages so that the message with all the piece metadata is at most of the MAX_PIECE_SZ byte size. These messages of appropriate size are returned.

Getting the scheme ID and the next sequence number

The function scheme_id represents the unique identification of the currently used scheme and its parameters. The idea is that each piece carries this ID. Based on some agreed-on database of configurations, the receivers would not need to know the parameters or scheme configuration of the sender identity. They would be able to work with pieces independently. For example, the HORST scheme with $n = 256, k = 32, \tau = 16$ using ChaCha20 as PRNG and the Sha3-256 hash function would have an ID 1.

The next_seq returns the sequence number for the next piece to be signed and broadcasted and updates the number in the instance state.

Selecting a key to sign a piece

This subsection describes the semantics of a function called select_key that is used in the algorithm 1. Rather than in pseudo-code, it is described in general terms giving freedom to implement it as anyone might find appropriate. This function returns the key used to sign the next piece and updates the information

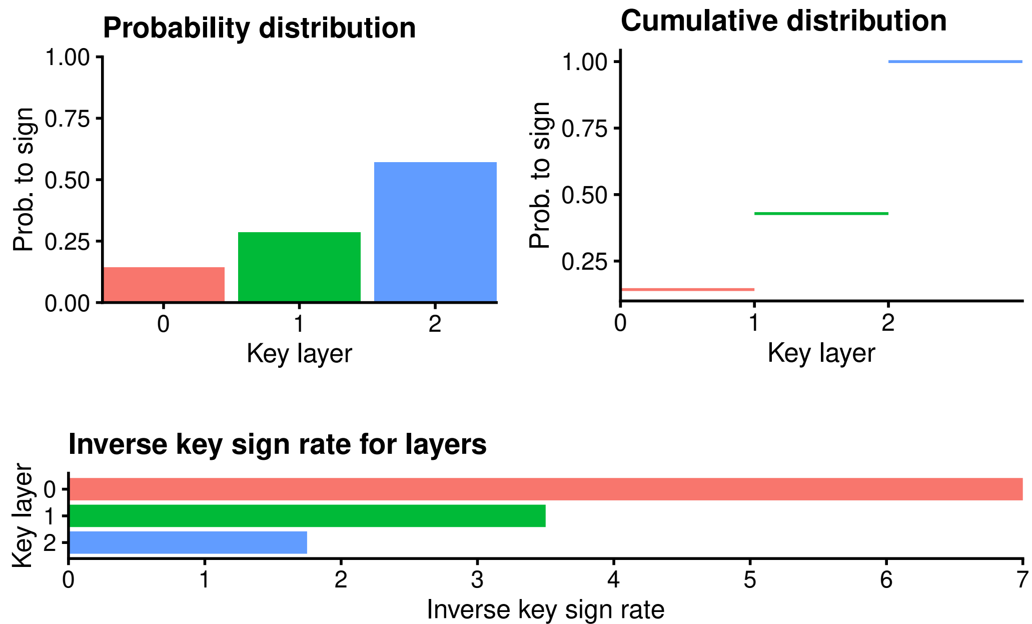


Figure 2.7 The example layer selection model for `KEY_RATIOS = [(4,0), (2,0), (1,0)]`. The probability of selecting the specific layers for signing the piece and cumulative distribution is shown. The bottom figure shows the inverse layer sign rate — i.e. how many pieces, on average, are signed before the given layer is chosen for signing.

about the key usage in the instance's state; most importantly, the maximum number of key usages cannot be exceeded. If a key is exhausted, the underlying few-time signature scheme must generate the new one.

With every piece to sign, the sender must determine what layer should be used for signing. This selection must adhere to the configured layer lifetime ratios defined by the `KEY_RATIOS` parameter. This list dictates the expected key lifetime ratios between the layers; the key lifetime is the number of signed pieces for which the given key on a given layer remains active, i.e. the key can still generate at least one secure signature. In this context, the time is measured in the number of signatures the sender generates, i.e. the number of signed pieces. So if one says, 'The key lifetime ratio between layer i and j is K.', it means that the layer j has to be chosen for piece signing K-time more often than the layer i. It is important to note that the number of signatures one key from the underlying few-time signature scheme can securely issue does not affect nor is affected by this lifetime ratio; it remains constant for every layer. This lifetime ratio just determines how often the specific layer is selected to generate the signature.

This behaviour is achieved by constructing a discrete probability distribution

based on these lifetime ratios and then sampling what layer to use. To be specific, the example parameter `KEY_RATIOS = [(4,0), (2,0), (1,0)]` is used; the example is visualised in the figure 2.7. The syntax contains a list of tuples; the first element in the tuple specifies the *expected lifetime ratios*. The second element in the tuples is a *key pause* — this will be discussed later in section 2.2.2. If the list does not contain tuples in a non-increasing sequence, it is sorted first; the first element should define the longest-living key layer. To get from the lifetime ratios to probabilities of signing with each layer, one inverts the ratios and normalises them by their sum; this way, they sum up to one and can form a discrete probability distribution. To formalise this, for the sequence of lifetime ratios $T = (t_0, \dots, t_n)$ one computes $T^{-1} = (t_0^{-1}, \dots, t_n^{-1}) = (\frac{1}{t_0}, \dots, \frac{1}{t_n})$ and the sum $S = \sum_{i=0}^{n-1} t_i$ and finally

$$P = (p_0, \dots, p_n) = (\frac{t_0^{-1}}{S}, \dots, \frac{t_n^{-1}}{S}).$$

In this thesis, the expected number of signed pieces until signing with the key from the specific layer is called the *inverse layer rate*; since it is an inverse value of the probability that the layer is selected. In other words, the inverse layer rate R_i says that the i-th layer will, on average, sign every R_i-th piece. It also corresponds to geometric distribution — the likelihood of choosing the key on the layer i is p_i. Therefore the expected number of tries to sign the piece with this layer for the first time is $\frac{1}{p_i}$. To compute the inverse layer sign rates, one computes the expected values for geometric distribution with parameter p_i as follows:

$$R = (r_0, \dots, r_n) = (\frac{1}{p_0}, \dots, \frac{1}{p_n}).$$

With this in hand, one can compute the expected absolute number of signed pieces the given layer key can stay alive. Let's assume the underlying scheme allows securely signing up to 20 messages with one key pair. One does multiply the inverse layer rate by this number and gets the average lifetime of a layer key in the absolute number of signed pieces. With our example setting, visualised in the figure 2.7, we see that the L_0 key will, on average, live through 140 signed pieces.

The `KEY_RATIOS` along with the `PRE_CERT` parameters are used to configure the protocol to allow receivers to miss a certain number of pieces while keeping the ability to re-authenticate the original identity within some reasonable time. Generally, the data overhead increases if more significant piece misses are required with a lower time to re-authenticate for the receiver. This time to re-authenticate after a piece miss is discussed in more detail in the section 2.3.2.

Key scheduling The presented key selection procedure works fine when considering the average case. Since keys are sampled from probability, the key lifetimes may deviate from the average case. Consider a situation where the sender wants to give receivers a strong guarantee that if they miss at most the specified number of pieces, they will be able to re-authenticate. With the key selection described above, this cannot be guaranteed, but it only relies on the expected values of the geometric distribution. In extreme cases, the keys meant to live for a very long time can be exhausted very soon, and the receivers cannot re-authenticate after the expected number of missed pieces.

One can specify key pauses on each layer in the `KEY_RATIOS` parameter to avoid this situation. In the example in the figure 2.7, the second number in the tuples was zero; this is the key pause. The zero value denotes that no key pause is applied on the given layer. But consider `KEY_RATIOS = [(4,100), (2,50), (1,0)]`; the L_0 layer now has a pause equal to 100% of its inverse sign rate required. The L_1 is required to have a 50% pause. The pause specifies that the given layer cannot be used to sign a piece more often than the percentage of their inverse sign rate. To give a concrete example of pausing keys — looking at the mentioned example, the L_0 layer has inverse sign rate 7, and the pause should be 100% of this, that is 7 as well; this means that there must be at least 6 pieces signed with the different layer between every two pieces signed with L_0. With L_1, which has an inverse sign rate of 3.5 and 50% pause applied, it requires that there must be at least one signature with a different layer between every two pieces signed with this layer; because $3.5/2 = 1.75$ and the value is ceiled to a natural number.

Certification of public keys

The second critical principle that the protocol relies on is that each piece carries public keys to be certified for the receivers. This way, receivers know in advance what keys will be used in the future by the given sender identity. This pre-certification gives receivers enough room to miss a certain number of pieces without losing the ability to re-authenticate that the pieces are coming from the same identity when they start to receive them again. This functionality of choosing the public keys to be certified with the next piece is abstracted behind a function named `select_pks`. For the selected key to sign this piece, it returns an ordered list of public keys to be announced (i.e. certified) with the next piece. The public keys are prefixed with a byte that encodes the layer this key is from, as specified in the figure 2.4. The first public key must always correspond to the secret key that signs this piece. The rest of the public keys can be chosen in a way that allows for consecutive receive of pieces without losing the ability to authenticate some of them. It is important to note that also keys that were

already exhausted must be certified. The receivers need proof that the sender has the secret part for certified public keys. The sender proves this by certifying the key by the key that is already part of the identity and then by sending a piece signed by this key that certifies some node from the identity — i.e. this key is part of a strongly connected component of an oriented graph showing the relation 'key X certifies key Y'. This is thoroughly described in the section 2.2.3.

This thesis recommends two strategies where one provides the most robust solution while sacrificing a bit of data overhead and the second with less overhead but with a higher probability of failing to authenticate some pieces due to missing public key that was not announced in advance.

Rectangle certification strategy This strategy sends all public keys currently present in the hierarchy of the sender's layers. It is called rectangle certification because we certify the whole rectangle of keys that are currently present in the sender's layers. The keys present in the current layers are visualised in the figure 2.6; the keys from all three coloured sections — depleted, active, future — are certified with every piece. The size of pre-certification windows is determined by the parameter `PRE_CERT`; when combined with the number of layers used L, the number of public keys that will be attached to each piece is $\text{PRE_CERT} \cdot L$. Multiplying this with public key size from the underlying scheme yields the data overhead related to pre-certification.

Cross certification strategy This strategy includes all public keys that are currently active on every layer. Provided that there are L layers, it is L keys. Also, a random key from every future and depleted column are selected to prepare receivers to receive pieces signed by the subsequent keys. Looking at the figure 2.6 with `PRE_CERT` = 2, for example, certified keys for this strategy could be $(K_{0,0}, K_{1,0}, \dots, K_{j,0}, K_{1,-2}, K_{0,-1}, K_{0,1}, K_{0,2})$. The number of attached keys with this strategy is $L + 2 \cdot$ `PRE_CERT`. This strategy can lead to occasional authentication misses for receivers if some key is not certified often enough and can lead to longer re-authentication time because for the receiver to get all public keys from the current rectangle of keys in the layers can take longer time than in the previous strategy. On the other hand, this strategy saves data overhead.

Storing the sender state persistently

The state of the sender instance must be stored persistently each time some key generates a signature. The function `store_sender` has to write it to the provided file `ID_FILE` so that the function load_sender can load it correctly. This allows the sender to stop the stream and continue later. Moreover, the device the sender

is using may crash anytime; the instance must always be kept in a consistent state. The important note is that it is not acceptable to send the signed piece before the state is stored; if the device crashed after the piece was sent and before the state was stored, the old state would be loaded the next time, and the same signature would sign a piece with the same sequence number but with different data. This is a problem since some receivers may have already received the piece sent before the crash. The state must be stored before sending the piece over a network.

Generating a signature and assembling a piece

The function `length` returns the number of public key wrappers — public key prepended with its layer number — that is to be attached to this piece. As expected, the `size` function returns a byte size of the provided structure.

To generate a signature for a piece, the piece without scheme ID and the signature itself must be assembled into a contiguous block of bytes as specified in the figure 2.4. This block of bytes is then signed with the underlying signature scheme. The secret key selected by the `select_key` function is used for signature generation. The scheme ID and the signature are then prepended to the signed block. The piece is complete and is prepared to be fragmented and sent via the network interface.

Sending a piece via a network

Before sending a piece over a network, it must be split into fragments; the fragment format is defined in the figure 2.5. The function `to_fragments` splits the piece into fragments of such size that when the datagram header is appended to it, the total size is at most `MAX_DGRAM_SZ` bytes. Each fragment has a header consisting of a piece ID; it is a unique identifier that allows the receiver to group the fragments by it. Then the index states the start of the interval where the payload from this fragment belongs in the whole piece. The payload size of the fragment determines the end index. The most significant bit, named `more`, of this index indicates whether this fragment is the last. Therefore the starting index inside fragments can be at most 2^{31}.

To know to which receivers the datagrams should be sent, the function `active_subscriber` reads the addresses of active subscribers from the shared structure maintained by `subscribers_task`. The final step is to send it via the network interface — that is what the function `send_fragments` does.

Keeping a registry of live subscribers

The sender must keep a database of receivers that want to receive the data from it. The sender expects the interested receiver to deliver a periodic heartbeat —

a message containing the magic word `0xBEAD`. The sender listens for heartbeat messageson the IP address and port specified by the parameter `SENDER_ADDR`. Upon receiving such a heartbeat, the sender will consider this receiver alive for the time defined by the parameter `RECEIVER_LIFETIME`. If no subsequent heartbeat arrives, the receiver is considered dead and deleted from the active subscribers; this is what the `prune_dead_receivers` function does. The corresponding pseudo-code is displayed in the algorithm 2.

Algorithm 2 The semantics of the task of keeping the database of active receivers up to date. The datagrams with the predefined magic word are periodically received from the UDP socket. The source senders are considered active for a limited time, and pieces are sent to them.

```
function SUBSCRIBERS_TASK(s, addr)
    SH.sender_socket ← listen(addr)
    while s.running do
        if (heartbeat, peer) ← try_recv(SH.sender_socket) then
            if heartbeat == 0xBEAD then
                alive_until ← get_time() + s.p.RECEIVER_LIFETIME
                insert_update_receiver(SH.subs, peer, alive_until)
            end if
        end if
        prune_dead_receivers(SH.subs)
    end while
end function
```

2.2.3 Receiver role

A receiver role starts with generating or loading the existing sender identities from the identity file; a new receiver identity database is created if this fails. Then the receiver is bound to receive data from the provided sender.

Upon receiving datagrams with valid fragments, the receiver assembles them to complete pieces and verifies their authenticity by checking the signature and the signing identity. The attached public keys are inserted into the identity database, and the identities are updated accordingly; also, obsolete keys are pruned. The pieces are sent to active receivers if a distributor role is enabled. Suppose the receiver is set not to deliver the messages — this can be the case if one wants this node to serve only the distributor role. In that case, the messages are discarded. Otherwise, these are delivered to the user application, complemented by their degree of authentication and the sender's identity. The detailed procedure is described in the algorithm 3.

Algorithm 3 The algorithms show the two interface functions of a receiver role. The comment numbers correspond to those in the brief algorithm description at the beginning of this sub-section.

```
function NEW_RECEIVER(p) -> r
    r ← load_receiver(p.ID_FILE)
    if r == ∅ then
        pks_graph ← init_pks_graphs()
        r ← (p, pks_graph)
    end if
    spawn fragments_task(r)                              ▷ Background task.
    spawn pieces_task(r)                                 ▷ Background task.
    spawn heartbeat_task(r, r.p.SENDER_ADDR)             ▷ Background task.
    if r.p.DISTRIBUTE != ∅ then
        spawn subscribers_task(p, r.p.DISTRIBUTE)        ▷ Background task.
    end if
    return r
end function
function RECEIVE(r) -> (r, message)
    while true do
        message ← dequeue_auth(SH.delivery_queues)
        if message != ∅ then
            return (r, message)
        end if
        message ← dequeue_unauth(SH.delivery_queues)
        if message != ∅ then
            return (r, message)
        end if
    end while
end function
```

The presented pseudo-code separates the receiver role into the main invocation of the `receive` function and three independent tasks running concurrently. It is important to note that it is unnecessary to implement it this way as long as the semantics are kept; this thesis chose this approach for the sake of readability and understandability. The `fragments_task` receives datagrams with fragments and is responsible for assembling correct pieces and passing them to the shared queue — `SH.pieces_queue`; the pieces from this queue are consumed by the `pieces_task`; it is responsible for authentication of the piece, update of the identity graph, sender state store, potential re-distribution (if configured with the distributor role as well) and for passing the message into the delivery queues with corresponding sender identity — `SH.delivery_queues`, if configured to deliver messages. The final delivery is described in the algorithm 3.

Delivering from the delivery queues

The algorithm 3 contains the functions `dequeue_auth` and `dequeue_unauth` to try dequeuing some messages to be delivered. There are two queues — one for messages certified or authenticated by the target identity and a lower-priority queue for unauthenticated messages.

The certified and authenticated messages must be delivered in a strictly growing order of sequence numbers. If there is a missing message with a sequence number next to be delivered, but the messages with higher sequence numbers are already in the queue, the `DELIVERY_DELAY` timeout is applied, and once it elapses, the next-in-line sequence number message is delivered. This in-order delivery and delivery timeout is hidden inside the `dequeue_auth` function.

Unauthenticated messages should be delivered in arbitrary order and with lower priority than those authenticated or certified. The `dequeue_unauth` function handles the low-priority delivery in an arbitrary order.

Receiving and assembling of fragments

Based on a periodic heartbeat to the target sender, the receiver starts getting datagrams that should contain valid fragments; these must be grouped by their `fragment_ID`, and the final piece must be assembled based on the `offset` and `more` bit. When the complete piece is assembled, it can be pushed to the shared queue of complete pieces for further processing. The incomplete pieces should be discarded after `FRAG_TIMEOUT` elapses. This functionality is hidden inside the background `fragment_task` job. The pseudo-code for this is available in the algorithm 4.

Algorithm 4 The fragments background task periodically reads datagrams from the receiver socket and assembles the received fragments into whole pieces based on their piece IDs. Whole pieces are passed to the shared queue containing the pieces.

```
function FRAGMENTS_TASK(r)
    merger ← init_merger()                          ▷ Merges fragments to pieces
    while frag ← try_recv(SH.receiver_socket) do
        piece ← insert_frag(merger, frag)           ▷ Returns complete pieces
        if piece != ∅ then      ▷ If some piece was assembled with this fragment
            enqueue_piece(SH.pieces_queue, piece)
        end if
    end while
end function
```

Piece authentication and identity management

Whenever there is some piece in the shared piece queue, the task responsible for piece processing dequeues the piece and starts the authentication process — this is written in pseudo-code in the algorithm 5. The piece is then parsed, and it is checked that the key for signature verification is at least certified by the target identity; if it is not, the message is delivered with an `Unauthenticated` flag. If the key is certified by the target identity, the identity is updated with the newly received public key — the process is described in detail in the section 2.2.3. Then the receiver's state is stored permanently to allow continuation in case of going offline or unexpected device fault. After that, the receiver checks if the signing key is part of the identity; if yes, the authentication state is elevated to `Authenticated` with the target identity. The next block is related to the distributor role. The semantics of the functions in the if block conditioned by the `DISTRIBUTE` parameter is the same as in the case of the sender; it is used in the algorithm 1 and described in the section 2.2.2 and the part about keeping the registry of active subscribers is discussed in the section 2.2.2. Finally, the message, sequence number, and authentication state are enqueued into shared delivery queues, from where they are dequeued according to the rules described in the section 2.2.3.

In the code, some functions should be specified as well. Starting with `target_sender_id` — this function returns the identity of the target sender based on the parameter `SENDER_ADDR`. Moving on to `parse_head`, this function splits the headed piece (`hpiece`) into scheme ID, signature and headless piece (`piece`). Then the `parse_piece` parses the sequence number, attached public keys and the message from the headless piece. The next if block is described in the section 2.2.3 — it handles the first ever piece received from this sender identity.

The functions `is_certified_by` and `is_auth_by` are self-explanatory — they check if the provided public key is certified or if it is part of the provided identity. The validity of the signature for the given message and given public key is checked by the underlying signature scheme with the function `verify_signature`. Then the most important part of the receiver role — updating the identity graph with the provided certified public keys; abstracted by the function `update_identity`, it is described in great detail in the section 2.2.3.

Algorithm 5

```
function PIECES_TASK(r)
    while hpiece ← dequeue(SH.pieces_queue) do
        sender_id ← target_sender_id(r)
        (scheme_id, sig, piece) ← parse_head(hpiece)
        (seq, pks, message) ← parse_piece(piece)
        pk ← pks[0]
        if is_id_empty(sender_id) then
            insert_id(r, sender_id, pk)
        end if
        auth_state ← Unauthenticated
        if is_certified_by(r, sender_id, pk) then
            if verify_signature(r.FTS, piece, sig, pk) then
                auth_state ← Certified(sender_id)
                update_identity(r, sender_id, pks)
                store_receiver(r)
                if is_auth_by(r, sender_id, pk) then        ▷ If this key is identity
                    auth_state ← Authenticated(sender_id)
                end if
                if r.p.DISTRIBUTE != ∅ then
                    frags ← to_fragments(hpiece, s.p.MAX_DGRAM_SZ)
                    subs ← active_subscribers(SH.subs)
                    send_fragments(subs, frags)
                end if
            end if
        end if
        if r.p.DELIVER then
            enqueue(SH.delivery_queues, message, seq, auth_state)
        end if
    end while
end function
```

Trust on first use The protocol needs one important assumption to prove that the data received are from the authenticated senders. We need to trust some identity at some point and mark it as trustworthy. The usual centralised approach is to use some authority to verify the identity behind a key and certify it. In this case, the receiver trusts the first message from the target sender that comes — in the pseudo-code, it is the ‘is_id_empty�‘ function; this is the main principle used in TOFU scheme [38]. Thus we need to assume that the sender of the first message is the sender from whom we want to receive data. After that, we can prove that the subsequent messages were sent by the identity we trusted initially (or sent by a group of trusted senders).

Updating the identity graph Keeping the identity graph is crucial for the receiver’s role; the function `update_identity` does this in the pseudo-code. This graph makes it possible to authenticate the data stream from already-known identities, even with significant message losses. The key principle is that after trusting the first message, the rest of the messages should be authenticated thanks to a constant flow of certified public keys from piece metadata. These attached keys will be used in the future, and those already depleted will never sign new pieces. Certification of exhausted keys is very important because, thanks to that, the receiver can prove that the sender also has the secret part of the key. Accepting a piece with some public keys does not prove that the sender owns these keys. In fact, an adversary can try to certify someone else’s keys to trick the receiver into trusting it.

To prove that the sender owns the key, it must be not only certified by it, but also this key must sign some piece, and that piece must certify some of the keys from the currently held identity. In terms of a directed graph and the relation: ’The key X certified the key Y.‘, this condition states that the public key in question must be in the same strongly connected component as the identity. The figure 2.8a further illustrates this on an intermediate state of the receiver’s identity graph. Nodes are public keys that were certified by some identities that the receiver was receiving from. We can see that the receiver knows two distinct identities — denoted by red and green nodes. The key k_4 is not yet part of any identity, but both red and green identities have certified it. The are three possible reasons why this happens:

1. The green identity is an attacker and falsely certified k_4 even if it does not own the key. The resolution is visualised in the figure 2.8b.

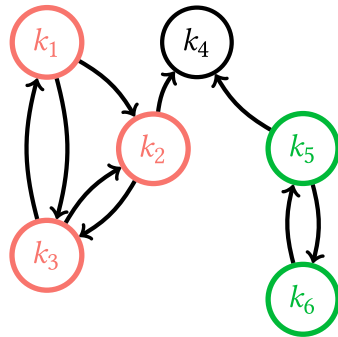
2. Possibly the red identity has falsely certified k_4. The following situation for the receiver looks like this figure 2.8c.

3. Finally, the receiver might have received the data from the same identity but using a different name or distributor. In this case, green and red identities are the same; The result is shown in the figure 2.8d. The final colour is a mixture of those two colours.

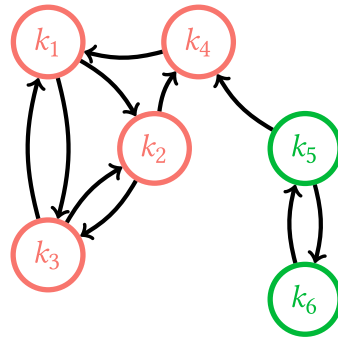
Pruning of obsolete public keys To avoid an infinite accumulation of public keys in receivers, the keys that are proven to be obsolete already can be deleted. An obsolete key will not be certified by the sender anymore. Based on the pre-certification size the sender uses, one can compute when it is safe to delete keys on any layer and which one to delete. The pre-certification size (PC) is determined by the number of public keys attached in pieces. Also, the receiver must mark each public key it certified by the sequence number it was first certified; this will be used to determine what keys to delete. This behaviour is part of the `update_identity` function.

Based on pre-certification window size, for a given layer, if there are more than $2\text{PC} - 1$ keys, delete the keys with the lowest sequence numbers of pieces they were certified. The situation is illustrated in the figure 2.9 as follows: Let's focus on the key k_i. How many keys with higher sequence numbers on the given layer must the receiver have to know that the sender will not certify this k_i anymore? Because the order or processing pieces can be arbitrary — due to the underlying network interface —, one must account for the worst-case scenario — let's assume that the k_i was received as the first one while the key $k_{i-\text{PC}}$ was active — i.e. was signing the pieces. This is the first time the k_i key could have been certified for this receiver. Now, this key will be useful through all the red and green keys are active. But it is guaranteed that the keys with a green border are first certified with a higher sequence number; because the earliest the k_{i+1} key is certified is while $k_{i-\text{PC}+1}$ is signing pieces. This ordering of sequence numbers for keys also holds for all subsequent green keys. Consequently, the maximum number of keys on any given layer, where it is possible that even the one with the lowest sequence number can be useful, is $2\text{PC} - 1$. If there are more keys, order them by their sequence number and delete the lowest ones.

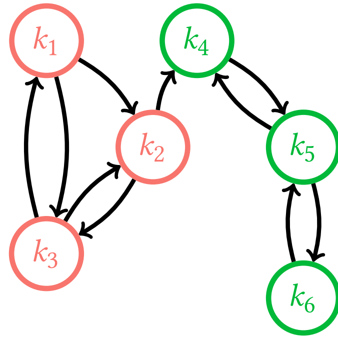
Losing the trusted identity After a receiver starts receiving messages from a sender, it begins collecting the certified public keys that the sender sends along with the messages. Based on the protocol configuration, these keys will grant a re-authentication if the receiver misses less than some number of messages; in other words, the receiver can go offline and come back, and the sender's authenticity will be renewed without the need to trust the first message blindly. However, suppose it misses a certain number of messages. In that case, the protocol won't be able to verify that the messages are coming from the same identity, even if that



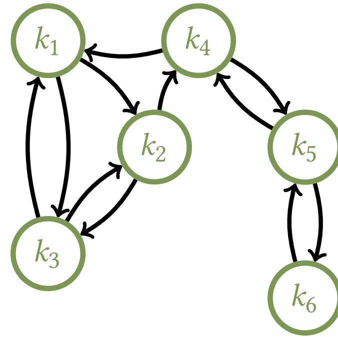
(a) The two possibly distinct identities have certified k_4.



(b) The piece signed with k_4 and certifying k_1 has been received.



(c) The piece signed with k_4 and certifying k_5 has been received.



(d) The piece signed with k_4 and certifying both k_1 and k_5 has been received.

Figure 2.8 A visualisation of different situations as they may evolve depending on a different piece received. The identity nodes are emphasised with coloured nodes — red is one identity, green is the other, and the mixture of those colours — the khaki one — is the merged identity of red and green. Any given node can belong at most to one identity; this assumes that the sender's secret keys were not compromised. The **(a)** subfigure sketches the situation after receiving from the two differently named sender identities. In **(b)**: If a piece signed with k_4 is received and it certifies some of the keys from the identity, it becomes part of that identity. In this case, k_4 certified k_2 and became part of the red identity. The situation **(c)** is similar to (b), only the green identity has proved it owns the secret key to k_4. The situation **(d)** shows the state after two so far distinct identities were proved to be the same; this happened because a piece signed by k_4 arrived and certified both k_1 and k_5.

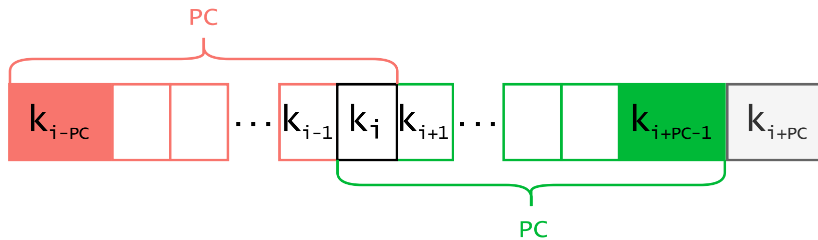


Figure 2.9 A key layer showing when the key k_i may have been first certified — when the key $k_{i-\text{PC}}$ was active; and when this key is proved obsolete — when the key $k_{i+\text{PC}}$ is active. Thus, one can delete keys with the lowest sequence number of the first certification if there are more than $2\text{PC} - 1$ keys on any layer. 'PC' is a shorthand for the `PRE_CERT` parameter, which stands for pre-certification size.

was the case. This behaviour is the implication of the protocol design because the receiver has missed too many messages that there are only obsolete keys in the receiver's identity graph. In other words, the public keys it has in its identity graph are no longer used for signing new messages.

However, two solutions that mitigate this problem can be used:

1. The sender can set up the public key hierarchy so that if the receiver receives the longest-living key, it is impossible to miss that many messages (because no computer could potentially broadcast that many messages in a reasonable time). This long-living layer, however, does not guarantee that the re-authentication time will be short in all cases; this must be achieved by finding the right `PRE_CERT` parameter in combination with the `KEY_RATIOS` parameter; this is discussed in the section 2.3.2.

2. The second solution is a system where the sender declares that it will not send more than some number of messages per unit of time. Combined with key pauses described in the section 2.2.2, one can compute the minimum guaranteed time that will conclude in re-authenticating the sender's identity. Again, this does not guarantee that the time to re-authenticate will be short; this is discussed in the section 2.3.2.

False certification and identity merge The protocol design relies on distributing the public keys used for signing data in the future. With this pre-certification mechanism, senders can verify the signatures of incoming messages if they have already accepted some of the certified keys for the given identity of the sender. The problem is that another identity may have certified the same public key; this can be seen in the figure 2.8a. This 'another identity' can be an attacker who sends a message to the receiver signed by the attacker's secret key but certifies

the original sender's public keys. Such false certification may be an attempt to merge with someone else's identity so the attacker can send the receiver messages appearing to be from the original legit sender.

This false identity merge is not possible due to the protocol design section 2.2.3; the exiting identity is extended by the new node if and only if it belongs to the same strongly connected component in a graph describing the relation 'Key X has certified key Y'. An attacker may be able to certify the original sender's key, but is not able to send a piece signed by the secret key of this falsely certified key that would certify the attackers public key. Only the original sender wields the secret key and will never sign the attacker's public keys; unless the sender wants to.

Sending heartbeats to the sender

To receive any data from the sender, periodical heartbeats containing the magic bytes `0xBEAD` must be delivered to it. The `SENDER_ADDR` parameter determines the address of the target sender; the format is an IP address with a port in the standard format followed by the petname for this sender. The petname serves the end-user to distinguish the already-known identities by a familiar and readable name. As mentioned earlier, if the receiver receives messages from the same identity under different names, the identity will merge and have both of the provided petnames.

The background task named `heartbeat_task` does that; the pseudo-code is to be seen in the algorithm 6. The sender parametrisation determines the required frequency of heartbeats — by its parameter `RECEIVER_LIFETIME`, but the frequency should be higher due to possible datagram losses. The time between each heartbeat sent is configured by the parameter `HEARTBEAT_PERIOD`. If the heartbeats get to the sender with lower frequency, there will be some message losses because the sender will consider this receiver dead for some time intervals.

Algorithm 6 The receiver must periodically send the heartbeat containing the magic byte sequence so the sender sends him signed pieces.

```
function HEARTBEAT_TASK(r, addr)
    while r.running do
        magic ← 0xBEAD
        send(SH.receiver_socket, addr, magic)
        sleep(r.p.HEARTBEAT_PERIOD)
    end while
end function
```

2.2.4 Distributor role

Even though the distributor role is mentioned as stand-alone, it is, in fact, a role that is implemented inside the receiver role. The distributor role is enabled for the receiver if the parameter `DISTRIBUTE` is set. It contains the IP address and port where this receiver will be listening for heartbeats from receivers. Suppose the receiver should distribute the certified and authenticated pieces from the identity that is known to it. In that case, sending the pieces to active receivers after these have been authenticated is enough. The distributor shall not modify the pieces. One can see that included in the receiver task in the algorithm 5 and also in the algorithm 4, where the task for keeping the active receivers (see the algorithm 2) is spawned — in this case, the address where it is listening is taken from the `DISTRIBUTE` parameter rather than `SENDER_PARAM`.

It is also worth mentioning that the receiver can be run in a mode where it does not deliver any messages; this is achieved by setting the parameter `DELIVER`. Combined with the `DISTRIBUTOR` parameter, one can run just the distributor node.

2.3 Parameter choice trade-offs

Throughout the protocol specification, multiple parameters were introduced. In this section, these are summarised, and their significance is assessed, especially concerning security and overhead trade-offs. Firstly, the overview and importance are stated for every parameter (each parameter is marked with labels 'S' and 'R' depending on what roles use it); this is followed by the categorisation based on the roles of the protocol. Then the relationship between protocol security and data overhead is discussed. The section is wrapped with a discussion about the re-authentication time based on the parameters.

`FTS (S, R)` The most important parameter of all is named `FTS`, which stands for a few-time signature scheme used as the underlying scheme. Based on the scheme used, the parameters differ. For the HORST scheme, it is n, k, τ and `HashFunction, CS PRNG`, for example. The scheme used, and the parameters determine the bit security, size of a signature and sizes of both secret and public keys. Choosing the fitting few-time signature is critical for the well-tuned protocol configuration — the example few-time signatures are described in the section 1.3.2. The used signature and its bit security usually have the biggest impact on data overhead since the signatures are of significant size.

`KEY_RATIOS (S)` The list of tuples specifies the number of layers and the ratio of their lifetimes and additionally specifies the key pauses on keys on

layers. This parameter is described in the section 2.2.2 and the key pauses in the section 2.2.2. Key ratios impact the maximum number of missed messages, the time it takes the receiver to re-authenticate, and the data overhead; along with the PRE_CERT parameter, it determines how many keys are certified with each piece.

`PRE_CERT (S)` This parameter determines how many keys will be announced in advance before they become active. The same number of keys is also announced in the opposite direction — to allow receivers to build the identities with their strongly connected components. Increasing the pre-certification size increases the number of public keys that are attached to each piece. Since the public keys are usually pretty small, one can be liberal with PRE_CERT and KEY_RATIOS parameters. This parameter is discussed in the section 2.2.2, and the practical examples are in the section 4.1.

`MAX_PIECE_SZ (S)` The maximum piece size sets the upper limit on how large pieces can be. If a user tries to broadcast data that would not fit into one piece, including the piece overhead, it is split into more pieces. One must consider that splitting piece in many datagrams generates additional overhead due to the fragment header — that is, 12 bytes per fragment.

`MAX_DGRAM_SZ (S, R)` Maximum datagram size provides the ability to adjust the size of datagrams to fit the needs of a network where the protocol will be operated. For example, with the UDP protocol, the huge size of datagrams can cause problems in some network nodes where MTU is low, and the node refuses to fragment the datagram. Datagrams of size 1500 bytes should be generally fine with today's Internet infrastructure.

`SEED (S)` The seed used to seed the sender's pseudo-random number generator. This happens only when initialising the new sender instance; after that, the state is persistently stored. Due to security, the seed should be of sufficient bit size and generated with enough entropy.

`SENDER_ADDR (R)` The address of the sender — for the sender, it is where it listens for heartbeats; for the receiver, it is the address of the target sender. It is an IP address with a port number.

`ID_FILE (S, R)` The file path to a file where the state of either sender or receiver instance is persistently stored after the inner state changes.

`RECEIVER_LIFETIME (S, R)` The time interval when a receiver is considered alive after receiving a heartbeat. During this time, the sender sends the active receiver messages.

DELIVER (R) If set to false, the messages will not be delivered to the user application; this is useful if one wants to run the node only as the distributor role.

DISTRIBUTE (R) This is the optional parameter that may be left unset. If it is set, it expects the IP address with a port number specifying at what address this receiver will listen for heartbeats. It does a similar thing as `SENDER_ADDR` for the sender instance. But, for the receiver role, the `SENDER_ADDR` parameter holds the address of the target sender.

DELIVERY_DELAY (R) The delay that the receiver waits for the next message in sequence before delivering messages with the higher sequence number; this is a safeguard not getting stuck on an undelivered message for receivers.

FRAG_TIMEOUT (R) Time after which the incomplete pieces are discarded; this happens whenever some datagrams get lost.

HEARTBEAT_PERIOD (R) The period in which the receiver sends heartbeats to the target sender; this should be significantly smaller than the target sender's `RECEIVER_LIFETIME` parameter.

2.3.1 Security vs. signature size

The primary goal of this protocol is to be secure, even against quantum attacks. Thus the most important trade-off one must decide when using the proposed protocol is between data overhead and security. The underlying signature scheme determines the security of the protocol. It is not surprising that usually, it is the signature size that contributes to the overhead the most. Unfortunately, quantum robust signature schemes tend to have larger signatures than traditional ones like DSA or ECDSA. The usual size of signatures from hash-based few-time signature schemes lies between 2kB and 40kB, based on the required bit security.

The attached public keys are the second important contribution to overhead; therefore, the size of public keys for the used scheme is critical. Using a scheme with small public keys greatly reduces the overhead because, based on the public key selection strategy, the piece contains either NUM_OF_LAYERS·(2PRE_CERT+1) or NUM_OF_LAYERS + 2PRE_CERT keys. Fortunately, various schemes have small public keys; usually, the size of a public key is equal to the hash function digest size.

The recommended procedure to find proper parameters for one's use case is to set the target bit security and select the signature scheme with its parameters that satisfy the security requirement. Selecting the scheme yields the signature size (`SIG_SIZE`) and public key size (`PK_SIZE`). The next step is to design the key

layers for the sender with pre-certification size (PRE_CERT) for the corresponding number of key layers (NUM_OF_LAYERS); these two determine how large message losses can be while keeping the ability to re-authenticate with the existing identity. These two parameters also impact how long it will take receivers to re-authenticate after a certain number of missed messages.

With these parameters, one can compute the overhead for one piece using the piece format specified in the figure 2.4. The public key selection strategies are described in the section 2.2.2. For the rectangle strategy, it is:

$$\mathrm{OH}_{\mathtt{rect}} = 24 + \mathrm{SIG_SIZE} + \mathrm{PK_SIZE} \cdot (\mathrm{NUM_OF_LAYERS} \cdot (2\mathrm{PRE_CERT} + 1))$$

For the cross strategy, it is:

$$\mathrm{OH}_{\mathtt{cross}} = 24 + \mathrm{SIG_SIZE} + \mathrm{PK_SIZE} \cdot (\mathrm{NUM_OF_LAYERS} + 2\mathrm{PRE_CERT})$$

It is important to note that some overhead is generated by fragmenting the pieces so they can fit into datagrams. The fragment format is described in the figure 2.5. The smaller the MAX_DGRAM_SIZE, the larger the overhead generated by the fragmentation. The total size of the overhead depends on the size of the actual message. The overhead, in general, is 12 bytes per fragment.

The concrete parameters and the results using the HORST signature scheme and the attached proof-of-concept implementation are shown in the section 4.1

2.3.2 Re-authenticating the prior sender identities

The important property of the proposed protocol is that once the sender misses some messages, it may take some time to re-authenticate that the messages come from the same identity. This time to re-authenticate is due to the limited number of signatures one key of the underlying scheme can generate. While the receiver is missing the messages sent by the sender — for example, while offline, the sender may exhaust some keys on certain layers and generate new ones, which this receiver does not know about. Consequently, when the receiver starts getting pieces from the same sender, it depends on when the sender signs with some key that this receiver knows about. There may be some messages that the receiver is not able to authenticate. Once the sender signs with a key that the receiver has in the identity graph, re-authentication happens. From that point, the subsequent messages will be authenticated again if using the rectangle public key selection strategy.

It is an important drawback in comparison to the signature schemes with ephemeral key pairs, but the reward for this is a quantum-resistant protocol for

distributed data broadcasting. As this is an important aspect of the protocol and the parameters determine how significant this drawback will be for the specific use case, the practical measurements and theoretical approximations are also presented. The reader can find them in the section 4.2.

Chapter 3

Proof-of-concept implementation

This whole chapter is dedicated to a thorough description of software that is part of this thesis — a library implementing a variant of the proposed protocol; it is delivered as a library for the programming language Rust (a Rust crate in the language's terminology). The following section will describe the second part of the software delivered with the thesis — an end-user application for live audio broadcasting. This application demonstrates how the library can be used for real-time data streaming and showcases the usage of the protocol in a use case that is well known to us all — radio broadcasting. Both the library and the broadcast application are implemented using Rust.

Rust essentials Because the attached software is implemented in Rust, a language not yet known by the community, this section will discuss a few important terminology terms and concepts that will help the reader comprehend the following chapter. It is also important to note that for brevity, some terms in widely-known contexts will be mapped to Rust terms even though they are not exactly the same; as an example, the reader will read that Rust traits are alternative to interfaces in C-like languages like C#. This mapping does not necessarily mean that these concepts are the same; in fact, Rust traits are more general than interfaces. Take these mappings with a pinch of salt and not as semantic equality.

Crate A crate is a Rust term for library — an encapsulated piece of code that explicitly exports constants, types, functions, structures and traits.

Trait Traits are fundamental concepts used in Rust. For the sake of this thesis, one can imagine them as interfaces in C-like languages like C# and concepts in C++20. A trait defines a set of methods a type must provide, and compilers ensure that all the trait bounds required by the generic types are satisfied.

3.1 HAB: A library for Hash-based Authentication Broadcasting

The HAB library — a crate in Rust terminology — offers a very simplistic interface. It is almost identical to the interface described in the section 2.1.5. It exports two structures — `Sender` and `Receiver`; both expect generic type parameters specifying the few-time signature scheme. Additionally, their constructors expect protocol parameters. These generic parameters and arguments to constructors are how to configure the protocol — for example, the size of pre-certification or distribution of the key selection for signing. After instantiating the sender or the receiver structure, one can start to broadcast via the method `broadcast`, or one can start receiving the data with a method called `receive`.

This section is structured as follows: First, the thesis looks at the library interface; all the methods are described with their parameters and outputs. The next step contains a high-level overview of the library's internal parts, roles, and responsibilities. This overview gives the reader a solid understanding of the library structure. Thus, the next subchapters closely look at these internal parts; these are implemented as Rust structures and functionally, these are alternatives to classes with static (in Rust called associated) and member functions. This thesis refers to these internal parts under a general term component. The section concludes with some limitations and drawbacks of the implementation and also discusses the future work to be done to point the library towards a production-ready library.

3.1.1 Overview of the HAB interface

The protocol has three roles in its design; the library exports only two role structures — `Sender` and `Receiver`. The distributor role is an optional part of the receiver role. The usage of the library starts in instantiating the correctly configured `Sender` of `Receiver` and then calling the one function they offer. One can refer to the file `lib.rs` in the source codes to see what identifiers are publicly exported; each statement starting with the keyword pub is publicly visible to the code using the library. The declarations of these types look like those in the listing 1.

The library is implemented generally; the exported `Sender` and `Receiver` types expect one generic type parameter — the type named `SignatureScheme` implementing the trait `FtsSchemeTrait`. In other words, a type that is used for key generation, signature generation and verification. One can include the trait `FtsSchemeTrait` from the library, implement it for his signature scheme, and use it with the library.

Listing 1 Type declarations of the main structures of the library.

```
// sender.rs
struct Sender<SignatureScheme: FtsSchemeTrait> { ... }
// receiver.rs
struct Receiver<SignatureScheme: FtsSchemeTrait> { ... }
```

Listing 2 A type declaration for the bundled-in HORST signature scheme.

```
// horst.rs
pub struct Horst<
  const N: usize,
  const K: usize,
  const TAU: usize,
  const TAUPLUS: usize,
  const T: usize,
  CsPrng: CryptoRng + SeedableRng + RngCore + /* ... */,
  HashFn: Digest + /* ... */,
> { ... }
```

For convenience, the HAB crate offers a HORST scheme out of the box. It is exported as the `Horst` type; naturally, it implements the `FtsSchemeTrait` trait and can be used as a type parameter for `Sencder` and `Receiver`. As expected, the signature scheme is Parametrising the re-authentication delayed as well; therefore, the `struct Horst` expects multiple type and non-type generic parameters; these correspond to the parameters described in the section 1.3.2. To take a closer look, the declaration of the type looks like in the listing 2.

The non-type parameters start with the keyword `const` and are followed by their name, and the type is declared after the colon. The `usize` type in Rust corresponds to C++ `size_t` type. The generic parameters named `N`, `K` and `TAU` are just as defined in the scheme in the section 1.3.2. `TAUPLUS` is just $\mathrm{TAU} + 1$, `T` is just 2^{TAU}. Regarding the type parameters, `CsPrng` is a cryptographically safe pseudo-random number generator as discussed in the section 1.2.2 and `HashFn` is a type that is to be used for hashing the message and inner Merkle tree — an implementation of cryptographic hash functions is discussed in the section 1.2.1. In the syntax, identifiers after a colon in case of generic type parameters are traits required for the type to implement. In this case, the type for CSPRNG is required to implement three traits. In simple terms, they require the type to be cryptographically safe, be seedable and provide the usual RNG interface. The hash function is required to implement the trait `Digest`; it means that the hash function must be the one with fixed-size output and implements the standardised hash function interface.

The reason for having two generic parameters that are easily derived from the others is that the current version of stable Rust does not allow arithmetic on generic parameters in places where compile-time constants are expected. This feature is already in the nightly build; it will be possible in the future, but this thesis includes code compatible with stable Rust at the time of submission.

Using the instances

At this point, the reader can correctly fill in the generic parameters into the `Sender` and `Receiver` types. The next step is constructing their instances; this is where the protocol parametrisation comes in. The library exports two structure types that are just containers for the parameters — `SenderParams` and `ReceiverParams`.One imports them, fills them with the desired parameters, and uses them as parameters for the `Sender` and `Receiver` constructors. To be concrete, the listing 3 and the listing 4 are roughly how the declarations look like.

Most of the parameters have already been described in the section 2.3; only in the code are in the snake case. Considering the previously mentioned section and code comments, their meaning should be clear. The remaining parameters are just technical ones that are not important to know about.

As for the distributor role, one can set the `is_distributor` field in the `ReceiverParams` to `true`; with this on, other receivers can subscribe to this receiver as if it was the original sender of the data. The distributed data sending takes some load off the original sender.

Now, the sender and receiver instances can be created by calling their constructors with the corresponding parameters we have just described. Rust has no special behaviour regarding constructors; they are just associated (alternative to static) functions for the given type. By convention, they are usually called `new`. The function signatures are displayed in the listing 5.

The semantics should be pretty straightforward, but to be on the safe side — on structure `Sender`, an associated function called `new` is defined that takes an owning parameter `params` of type `SenderParams` and returns an instance of type `Sender`. For the `Receiver`, the semantics are analogous.

Finally, the instances are correctly instantiated and are ready to be used. Their interface is rather simple — each of them offers one method. The sender instance allows the user to repeatedly broadcast the data for active receivers by calling the method `broadcast`. Symmetrically, on the receiver side, the instance allows calling `receive` and blocking until a valid message is delivered. The interface is shown in the listing 5.

The `Sender` structure has a method named `broadcast` defined. Methods have the first special parameter containing the keyword `self` or some variant of it — `&self` and `&mut self`; this distinguishes associated functions from member

Listing 3 A declaration of the structure for sender parameters.

```
// sender.rs
struct SenderParams {
  /// A filename where the identity will be serialised.
  pub id_filename: String,
  /// A seed for the pseudo-random number generator.
  pub seed: u64,
  /// A distribution for key selection algorithm.
  pub key_dist: Vec<Vec<usize>>,
  /// Number of keys to certificate in advance.
  pub pre_cert: usize,
  /// Number of signatures one key can sign.
  pub key_lifetime: usize,
  /// A maximum byte size of one piece.
  pub max_piece_size: usize,
  /// A maximum byte size of one datagram.
  pub datagram_size: usize,
  /// A maximum time between two heartbeats from
  /// the given receiver.
  pub receiver_lifetime: Duration,
  /// An address and port where the sender will be
  /// listening for heartbeats.
  pub sender_addr: String,
  /// A flag that indicates if the application should
  //// run or terminate.
  pub running: Arc<AtomicBool>,
  /// An alternative output destination instead
  /// of a network (useful for testing).
  pub alt_output: Option<mpsc::Sender<Vec<u8>>>,
}
```

Listing 4 A declaration of the structure for receiver parameters.

```
// receiver.rs
struct ReceiverParams {
  /// A filename where the identity will be serialised.
  pub id_filename: String,
  /// Maximum time to delay the delivery of a piece
  /// if subsequent pieces are already received.
  pub delivery_delay: Duration,
  /// If this receiver should also re-send
  /// the received pieces.
  pub is_distributor: bool,
  /// The IP address of the target sender.
  pub target_addr: String,
  /// The name of the target sender (the petname).
  pub target_name: String,
  /// A flag that indicates if the application should
  /// run or terminate.
  pub running: Arc<AtomicBool>,
  /// An alternative output destination instead
  /// of a network (useful for testing).
  pub alt_input: Option<mpsc::Receiver<Vec<u8>>>,
}
```

Listing 5 The sender and receiver API.

```
// sender.rs
Sender::new(params: SenderParams) -> Sender
Sender::broadcast(&mut self, msg: &[u8]) -> Result<(), Error>

// receiver.rs
Receiver::new(params: ReceiverParams) -> Receiver
Receiver::receive() -> Result<ReceivedMessage, Error>

// common.rs
pub enum MessageAuthentication {
  /// The message was not sent by the target
  /// identity nor by identity certified by it.
  Unverified,
  /// The message was sent by the identity certified by the
  /// target identity (not proved to be the identity itself).
  Certified(SenderIdentity),
  /// It is proved that the target identity sent the message.
  Authenticated(SenderIdentity),
}
```

methods in Rust. This `broadcast` function accepts a mutable reference to the `Sender` instance to act on. This mutability implies that the call of this method likely mutates the inner state of the instance, just like a C++ method that is not decorated with the `const` keyword. The second parameter refers to the data to be broadcasted, a reference to an array of unsigned 8-bit integers — bytes. This method returns a `Result` type — this type is a wrapper for either of the two types provided as generic parameters. In our case, if the broadcast succeeds, nothing is returned; the `()` notation is an alternative to void in C++. On the other hand, if something goes wrong, the `Error` type is returned. This error contains a message with the reason; it implements the standardised trait `std::Error`[1], so it has the usual interface methods.

The `receive` function blocks the thread until some message from the target sender is delivered. Then it returns the `Result` type that contains the `ReceivedMessage` type or an `Error`. The received message is just a structure containing the message bytes, authentication state with the sender identity, if some, and a sequence number for this message.

The field `message` is just a vector of bytes, and `seq` is just an unsigned number stating this message's sequence number in the overall data stream. The `authentication` field is an enumeration type with three possible states; these states for each message are also shown in the listing 5.

3.1.2 Internal structure of HAB library

The library is structured into two important components already introduced in the previous part — `Sender` and `Receiver` structures with their generic parameter specifying the underlying signature scheme and protocol parameters provided as constructor parameters. These two are composed of sub-components; some are concrete structures, and some are abstracted out for potential substitution and rely only on specific interfaces. Also, some helper and general utilities are included and can be used by outside applications. The high-level overview of components and sub-components is displayed in the figure 3.1.

`Sender` structure

The `Sender` structure was already discussed a lot in the interface section because it is the publicly exported type that users of the library import; there is not much more to say about this sub-component. It uses `MessageSigner` and `NetSender` as its subcomponents.

The `Sender` fragments the whole data provided into the `broadcast` method into messages, so the total size of pieces is less than or equal to `MAX_PIECE_SIZE`

[1] `https://doc.rust-lang.org/std/error/trait.Error.html`

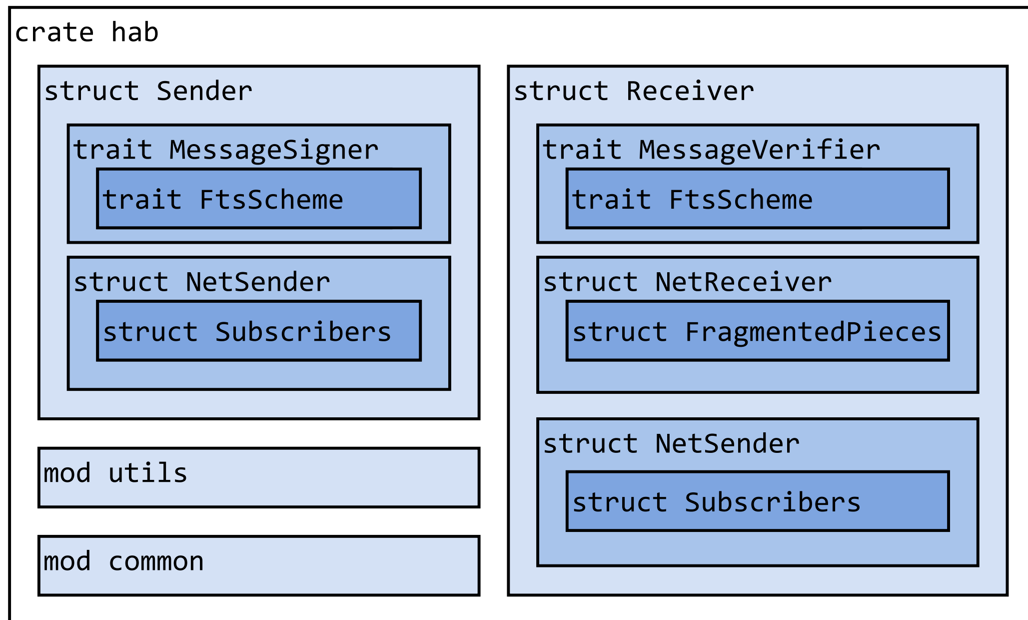


Figure 3.1 The schematic structure of the HAB crate, showing its main modules, structures and traits.

if needed; this means that if the provided data is too large to fit within one piece, it produces a sequence of pieces that are sent one after another. Besides that, it generates a new sequence number for the piece, asks the `MessageSigner` component to generate a piece for the provided message and then passes the result to the `NetSender` component that sends the piece to the actively subscribed receivers. The networking component also hides the logic behind keeping the database of active receivers; it is described in the corresponding section.

`Receiver` structure

Just as the `Sender` component, this was discussed as a part of the interface. The `Receiver` component uses `NetReceiver` and `MessageVerifier` as subcomponents. The constructor spawns a new thread; this thread is blocked until some piece is delivered by `NetReceiver`. The `MessageVerifier` component then performs the authentication check and passes the piece for delivery to the end user. Pieces are not delivered immediately because authenticated and certified pieces are delivered in order. If some piece is missing and the subsequent pieces are already ready for delivery, at most, `DELIVERY_DELAY` is waited before skipping the missing message. Pieces that are unauthenticated are delivered as they come. This behaviour is achieved by having one min heap for authenticated and certified pieces that keeps the next-to-be-delivered piece at the top. For other messages,

Listing 6 The trait declaration for stateful message signer.

```
// traits.rs
pub trait MessageSignerTrait {
    fn new(params: BlockSignerParams) -> Self;
    fn sign(&mut self, message: Vec<u8>, seq: SeqType)
    -> Result<Self::SignedMessage, Self::Error>;
    fn next_seq(&mut self) -> SeqType;
}
```

only a queue structure is used. Simply put, this thread is a producer of checked pieces that stores them in the shared queues.

The consumer for these authenticated pieces is the active loop inside the `receive` method. Whenever the user calls the `receive` method, it enters an active loop that periodically tries to dequeue some pieces for delivery. It is done so that the authenticated and certified messages have priority above those unauthenticated. There is no passive waiting here because the `receive` function must react whenever the `DELIVERY_DELAY` passes and subsequent pieces are ready.

When the parameter `IS_DISTRIBUTOR` is set to `true`, the receiver also forwards the verified pieces to receivers that ask this one to re-send them. The `NetSender` component is reused here; it works just as in the case of the `Sender` component. The difference is that this time, it is the `Receiver` component which asks the `NetSender` to broadcast the piece.

`MessageSigner` trait

A component implementing the `MessageSignerTrait` trait transforms a message into a piece; i.e. it chooses the key to sign it with, signs it, and determines public keys to be certified. It returns a type implementing `IntoFromBytes`; this trait implies that the type has methods `into_network_bytes` and `from_network_bytes`. Using these, one serialises the type into the array of bytes of the specified format (see the figure 2.4); this can be passed to `NetSender`.

Internally, the `MessageSignerTrait` type holds the state of CS PRNG and key layers and is responsible for key management. It then uses the underlying type with `FtsSchemeTrait` for issuing the signatures with the selected keys. A state of the BlockSigner must be persistently stored so the sender's identity is not lost on application termination. The trait is declared as displayed in the listing 6.

Altogether, the interface provides a constructor, a function for signing the message that returns the structured `SignedMessage` and can yield the next sequence number. The trait interfaces are briefly described because anyone can

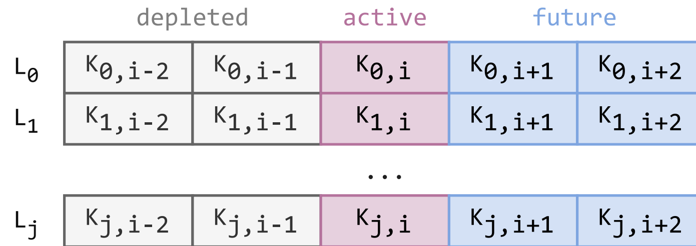


Figure 3.2 A structure that manages the key layers for signature generation and certification. This example shows the case with `PRE_CERT`= 2.

implement the traits for custom types and use them as drop-in replacements.

Key layers, key selection and certification The important part of the `MessageSignerTrait` implementation is managing the configured layers of keys. Our type `BlockSigner` implementing `MessageSignerTrait` takes a simplistic approach. A vector of double-ended queues is used; each queue is for the configured layer of keys. Each layer has the configured number of keys — in total, `2 * PRE_CERT + 1` keys are present on each layer. We have `PRE_CERT` depleted keys, one active and a `PRE_CERT` number of keys that are to be used in the future. The structure is visualised in the figure 3.2.

The discrete probability structure with weights corresponding to the configured key distribution is used to select what layer will be used for the signature. The key pauses are implemented as a sequence number of the previously signed messages attached to each layer. Whenever the layer is sampled, if it is too early, a new sample is taken until a different layer is selected. The sequence number is updated each time the layer is selected.

To select what keys will be certified in advance and what depleted keys will be attached, the strategy to attach all the keys from all the layers is used — in the protocol design, it is referred to as the "rectangle strategy". As the protocol requires, the public key that should be used for signature verification is attached as the first one.

`MessageVerifier` trait

`MessageVerifierTrait` is the counterpart to the trait `MessageSignerTrait`. In the HAB library, the incarnation of this trait is also the `BlockSigner` type; it implements both traits to share some pieces of code. This component is responsible for persistently holding the state of already-known identities with their petnames, for authenticity checks on received pieces and for inserting the newly received public keys into the identities. The trait interface looks like it is shown

Listing 7 The trait declaration for stateful message verifier.

```
// traits.rs
pub trait MessageVerifierTrait {
  fn new(params: BlockSignerParams) -> Self;
  fn verify(&mut self, piece: Vec<u8>)
    -> Result<VerifyResult, Self::Error>;
}
```

in the listing 6.

Expectably, it requires a constructor and a method that takes in a piece as an array of bytes and verifies it. The type returned by it — `VerifyResult` can be mapped to the `ReceivedMessage` that the external library API returns.

Managing the sender identities The public keys are kept as an oriented graph; nodes are public keys, and edges indicate relation "the key X certified the key Y". The keys that are part of a strongly connected component are considered the same identity; the Tarjan's algorithm for strongly connected components is employed [39].

Identity with the given petname is created upon receiving the first piece from it. The key used to verify this first message that the receiver must trust is the initial node with this new petnamed identity. Afterwards, this identity spreads as the strongly connected component grows and shifts. The strongly connected components are updated after public keys from each piece are inserted into the graph. It is important to note that as the protocol requires, only public keys from pieces that are at least certified are inserted. Unverified pieces do not mutate the receiver's state; they are just passed to the delivery queues with the `Unauthenticated` tag.

Thanks to the protocol design, the keys can be safely deleted after certain conditions are met. For the given layer, if there is more than `4 * PRE_CERT + 1`. This property is described in the section 2.2.3.

`FtsScheme` trait

This type must implement the trait `FtsSchemeTrait`, which is the incarnation of the few-time signature scheme used. The scheme must generate a new keypair, sign an array of bytes with the provided secret key, and verify the signature for the given array of bytes, the public key and the signature. The signature scheme trait must be stateless; therefore, the interface is designed as a set of static functions (in Rust, they are called associated functions). The trait interface is declared as shown in the listing 8.

Listing 8 The trait declaration of a signature scheme usable with the HAB library.

```
// traits.rs
pub trait FtsSchemeTrait {
    fn verify(msg: &[u8], signature: &Self::Signature,
    pub_key: &Self::PublicKey) -> bool;
    fn sign(msg: &[u8], secret_key: &Self::SecretKey)
    -> Self::Signature;
    fn gen_key_pair(rng: &mut Self::CsPrng)
    -> KeyPair<Self::SecretKey, Self::PublicKey>;
}
```

The type that implements the HORST scheme (see the section 1.3.2) inside the HAB library is `HorstSigScheme`. The implementation can be found in the `horst.rs` file. It is important to note that the provided implementation uses pregenerated secret keys, which means that all the random numbers are generated in advance and stored. As an optimisation, because these numbers are pseudorandomly generated, they can be computed from the seed on demand, but this implementation decided to take the precomputed path.

Functionality of `NetSender`

The `NetSender` component is responsible for sending the pieces to active receivers. To know to whom it shall send the pieces, it also listens for heartbeats from interested receivers and keeps the registry. Each receiver has the maximum lifetime RECEIVER_LIFETIME. If no heartbeat is received for this time, it is deleted and is no longer considered alive.

Because pieces can be relatively large, the `NetSender` component has to fragment it into fragments of the maximum size appropriate for the maximum UDP datagram size `MAX_DGRAM_SIZE`. The format of pieces is described in the figure 2.5. Then the datagrams can be sent to the active receivers.

The important thing to note here is that the send datagrams should be uniformly spread into acceptable time slots to reduce the possibility of queue overflow on intermediate devices. This datagram throttling is especially important when used via the Internet.

The second important note concerns receivers with IP addresses hidden behind the Network Address Translator [40]. In fact, these addresses are still quite common these days. For implementation to work with these types of addresses, the socket that receives the heartbeats must be the same as the one sending the data back. Without this, the backwards NAT mapping does not work.

Functionality of `NetReceiver`

The NetReceiver component is, as expected, the receiver counterpart to `NetSender`. It receives the datagrams from the network via socket, reads the fragments from the datagram, deserialises the fragment header and passes them to the helper structure called `FragmentedPieces`. Once some piece is complete, i.e. all the byte intervals have been received successfully, it is passed via a message queue to the `Receiver` component. There, a thread is blocked on a receive function on this queue to continue with the authentication process. In other words, in the context of this receiver pieces queue, `NetReceiver` is the producer, and the `Receiver` structure is the consumer.

`ActiveReceivers` structure

The `ActiveReceivers` component is implemented as a thread-safe binary tree map (alternative to C++ `std::map`) that can be shared safely between the threads. The key is a socket address of the receiver that requested the data by sending a heartbeat, and the value is a UNIX timestamp denoting the time until this receiver is considered alive. Whenever a new heartbeat comes, the UNIX timestamp is updated, or a new record is inserted. This `ActiveReceivers` structure is a wrapper class around atomically reference-counted reference holding a mutex that protects the binary tree map; this allows one to safely create references with shared ownership and pass them wherever needed.

In this case, one reference is used by the separate thread that processes the incoming heartbeats and updates the map of active receivers. The second is used for reading the active subscribers while sending the datagrams. The dead subscribers are also pruned when reading all the receivers to send the datagrams; if the given receiver has a timestamp past the current time, it is deleted.

`FragmentedPieces` structure

The `FragmentedPieces` structure is an intermediate storage for pieces not yet fully received. Pieces are fragmented into fragments, and datagrams carry these via a network; they may arrive in arbitrary order or not arrive at all due to the unreliability of the UDP protocol. Therefore this is a crucial part of the whole picture. Only complete pieces leave the `FragmentedPieces` structure. The inner state of the structure can be seen in the figure 3.3.

The format of fragments can be seen in the figure 2.5. It contains the ID of the piece and offset where the first payload byte fits in the whole piece buffer. Also, one bit indicates if this is the last fragment. The inner implementation is a `HashMap` where the key is the fragment ID, and the value is one fragmented block; the type is called `FragmentedBlock`. Each block keeps a preallocated vector of

ID2 END
ID6
ID1
ID0 END

Figure 3.3 An intermediate state of a structure responsible for assembling all the fragments for a piece with the given ID. The END markers are fragments with the `more` bit set to false. Blue byte intervals are present, and the white intervals are still missing. The format of fragments is described in the figure 2.5..

bytes and iteratively fills in the blank gaps as they come. It is important to note that the total size of a piece is now known until the fragment with the `more` bit set to `false` arrives; thus, the vector is resized on demand. To know what intervals are already filled and what are missing, the structure called `BufferTracker` is employed.

Tracking missing byte intervals The `BufferTracker` is implemented as a vector of already present intervals. If the new interval comes, it is inserted into the tracking vector; if the interval is adjacent to some existing intervals, they are merged into one interval by simply connecting their byte boundaries. The buffer is complete. There is only one interval starting from 0 and containing the end flag; byte intervals that come in with the `more` bit to `false` is marked, and this mark is also propagated through merging.

Exported modules `utils` and `common`

The exported modules work so the source code that imports them can use all public identified inside that module — types, traits, functions and constants. The `utils` module offers a variety of generally practical functions that are not bound to specific types. To give some examples — functions to compute hashes of byte arrays or to return current UNIX timestamps are offered. The `common` module contains several type definitions that may be useful when experimenting with the library — e.g. `SenderIdentity` or `MessageAuthentication`.

3.2 AudiBro: A real-time audio broadcasting

To demonstrate that the proposed protocol and the delivered HAB library are not just a nice thing unusable in actual use cases, the second part of the software

in this thesis is an example application that showcases the usage of the library for a simple real-time broadcasting software; both for sending and receiving. It is a minimalistic application for radio broadcasting over computer networks employing post-quantum authentication with large packet loss tolerance — the HAB implementing the proposed protocol leveraging the HORST scheme.

The section will introduce the usage of the application and will continue with the design description.

3.2.1 Interacting with the application

The application uses a terminal UI with two modes — broadcaster and a receiver. More about the application and how to use it is in the user documentation found in the appendix C.

Broadcasting with AudiBro

The application is launched with the flag `--sender` and the full address (the IP address and the port) from which the data will be broadcasted. As a broadcaster, there are a few options for audio files we can start broadcasting in real time. Or we can choose to broadcast the audio from the default audio input. With arrows, we can select the input source and hitting enter button starts to broadcast from the desired input.

Receiving with AudiBro

In this role, we start the application with the `--receiver` flag and provide the target sender to listen to (again, the IP address with the application port); in this case, it also includes the name of the sender, the so-called petname, to use the existing identities for authentication. As a receiver, we are presented a simple player where we can see the target name and address along with the current authentication state of the data we are being played — green as authenticated from the target identity, orange for only certified by the target identity and red for unauthenticated.

3.2.2 Design of the AudiBro application

This section discusses the important design aspects of the demonstrating application. It is designed as a terminal application with a simple terminal user interface.

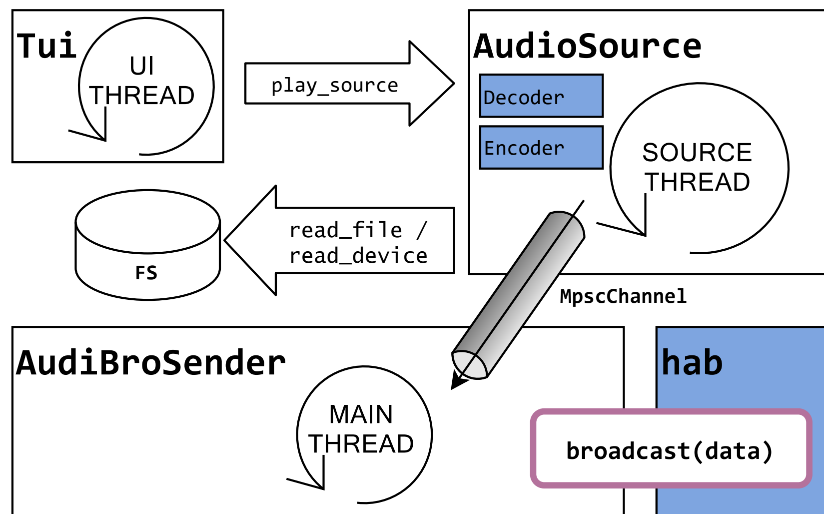


Figure 3.4 Overview of the AudiBro application running in sender mode. The interaction with the HAB library is also visualised.

Sender mode functionality

The sender component uses three main components — `Tui`, `AudioSource` and `AudiBroSender`. The application is built upon the hab crate; in the end, it is meant to demonstrate it in a real-world application. The design overview of the sender part is visualised in the figure 3.4.

The high-level flow of the sender role goes roughly like this: The user interface runs in a separate thread and periodically clears and writes the current state of the UI to the user. It reacts to keypresses of arrows that navigate through the input options, and the enter key is used to start broadcasting from the selected input. This input is passed to the `AudioSource` component, where the source thread runs. Based on the chosen input, it either loads the MP3 file from the disk or starts reading the default input device; an audio encoder and decoder are used. After playing a few seconds of the input audio and buffering it, the data is passed via `MpscChannel` to the main thread blocked on the channel until some data arrive. The main thread then calls the HAB library function `broadcast`, and the audio block in the form of a piece is on its way to active receivers.

Tui The interactive terminal UI is implemented using the `crossterm` crate that allows a simple way to implement interactive UI and capture the basic input while acting on it. The interface shows the statically defined set of available audio files and an option to stream from the microphone. When some input is selected, it is passed to the `AudioSource` component. The currently selected input method is marked in the UI with the colour.

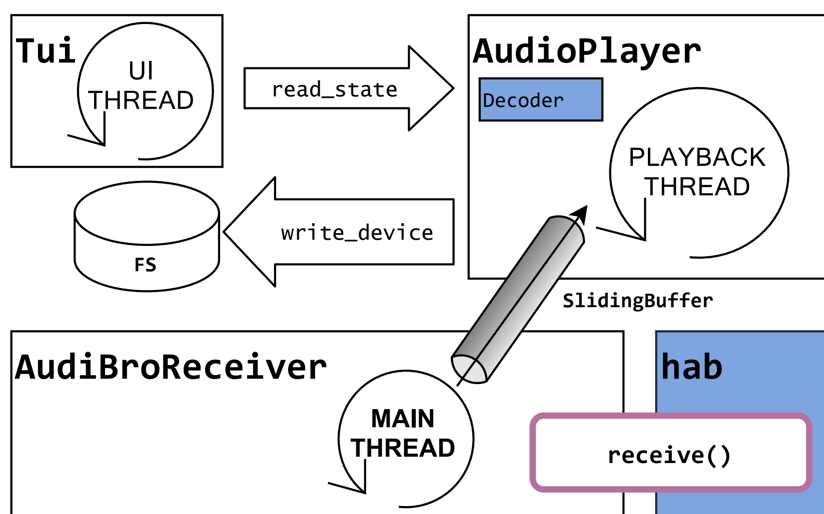


Figure 3.5 Overview of the AudiBro application running in receiver mode. The interaction with the HAB library is also visualised.

AudioSource A thread is running inside the component, waiting for some commands from the UI to come. Whenever a request to stream from an audio file comes, the file is loaded from the disk, and while using the decoder, we read MP3 frames. The crate `minimp3` is used as the encoder and decoder. Based on the sample rate, it is computed how many frames correspond to the configured number of seconds it should buffer before broadcasting it. The longer the buffer time, the bigger pieces can be; however, it generates a delay. If a request is to broadcast from the microphone, the samples are gathered and encoded with the encoder. In either case, the data block containing the MP3 frames is sent via `MpscChannel` to the `AudiBroSender` component.

AudiBroSender The role of this component is rather simple; its role is accomplished in the main thread. The thread is blocked on the `MpscChannel` until some stream of MP3 frames comes. It then forwards the data into the HAB library via the `broadcast` function.

Receiver mode functionality

The receiver comprises three main components — `Tui`, `AudioPlayer` and `AudiBroReceiver`. The receiver is, just as the sender, built upon the hab crate. The receiver design overview is visualised in the figure 3.5.

The overall cooperation of the components in the receiver is the following: On the receiver side, it starts with the function `receive` returning authenticated data;

this is where the HAB library comes in. The data is received by the main thread and is written into `SlidingBuffer`; this is a special buffer described below. The `AudioPlayer` component processes the data while employing the audio decoder and playing the audio to the default playback device. The user interface again runs in a separate thread, and the primary responsibility is to read the state of the audio source. Based on the state, it displays the target address and name; most importantly, it shows the current authentication state of the data that is played back to the end user.

AudiBroReceiver The `AudiBroReceiver` receives the pieces from the target sender with their authentication state and then pushes the data into a special buffer structure called `SlidingBuffer`. This buffer is a readable structure that allows a thread-safe push of the data on the one side — the receiver and at the same time, it is used as a buffer for the audio decoder on the other side of `AudioPlayer`. `SlidingBuffer` is a buffer where the buffered audio is kept; after it is played, the frames can be deleted. At the same time, new data is coming simultaneously.

AudioPlayer The `AudioPlayer` component consumes the data currently present in the `SlidingBuffer`. It plays the MP3 frames from the buffer employing the decoder, plays it via the default playback device and then deletes them. To play the compressed audio, the crate `rodio` is utilised.

Tui For the receiver mode, the UI is implemented the same as in the case of the sender. Only there are not many options that the user can do. The primary purpose is to display the authentication state of the played audio — if it is authenticated to be sent by the target identity, is it just certified by it, or is the data unauthenticated? This authentication state visualisation is achieved by periodically reading the state of the `AudioPlayer` component and then changing the printed output.

Chapter 4

Results and performance

The following chapter showcases the practical results of the proof-of-concept implementation with the HORST signature scheme and general recommendations based on theoretical approximations. Although the measurements were done using this specific instantiation of the protocol, the general algorithm to design parametrisation tailored for the particular use case and with an arbitrary signature scheme is presented; it is based on theoretical approximations supported by actual measurements with the PoC implementation.

This chapter is structured as follows: Firstly, the chapter presents the suggested HORST signature scheme parametrisations and a general approach for designing the protocol parameters for specific use cases. Additionally, the trade-off between security and overhead is described. The practical parametrisations are followed by the section that shows the theoretical estimations and empirical measurements of time to re-authenticate for the given numbers of missed messages. These measurements give the reader a clear understanding of using the protocol with any fitting scheme and for different use cases. Finally, the chapter concludes by presenting the trade-offs between security and data overhead; moreover, lessons learnt and future work are discussed.

4.1 Finding practical protocol parameters

To find a fitting set of parameters for the protocol, one must first know the specific use case where the protocol is to be used. Here, the general approach is presented using the HORST scheme and giving recommendations.

The most important aspect is security; at the very heart of the protocol, there is a few-time signature scheme. One should start with selecting a fitting quantum-robust signature scheme — such a scheme should focus on a small signature and public key size and runtime efficiency to fit also real-time applications. Since the

Instance name	Hash function	N	K	τ	Comment
HORST-RIPEMD-128	RIPEMD-128	16	16	8	low security
HORST-RIPEMD-128	RIPEMD-128	16	8	16	
HORST-RIPEMD-160	RIPEMD-160	20	10	16	
HORST-RIPEMD-160	RIPEMD-160	20	16	10	unvafourable signature size
HORST-SHA3-224	SHA3-224	28	16	14	unvafourable signature size
HORST-SHA3-224	SHA3-224	28	14	16	
HORST-SHA3-256	SHA3-256	32	16	16	
HORST-SHA3-384	SHA3-384	48	24	16	
HORST-SHA3-384	SHA3-384	48	16	24	unvafourable secret key size
HORST-SHA3-512	SHA3-512	64	32	16	

Table 4.1 The candidate signature scheme instances that were considered. The red candidates were found inappropriate due to insufficient security, unfavourable signature, or secret key size.

essence of the protocol lies in piggybacking additional public keys with every message, small signatures and public keys play a significant role in reducing data overhead. The chosen signature scheme must meet the target security. With a signature scheme, the maximum number of signatures one key can generate (i.e. key charges) is derived. Our example considers the candidate instantiations of the HORST scheme in the table 4.1. The specific parameters that are described in the section 1.3.2 are listed in the table. However, some of these are written in red, along with why they are unsuitable.

The candidate instances that are further considered are displayed in the table 4.2 with a public key, secret key and signature sizes. It is important to note that the HORST secret key size can be reduced significantly and traded for runtime speed by not precomputing it in advance but evaluating it on demand with PRNG and the seed. Since disk space is not an issue in this case, the runtime speed is preferred. Also, a classical bit security and the number of signatures that one key pair can generate without falling under the stated security are listed in the table.

Unfortunately, post-quantum signature schemes tend to have larger signatures, especially for higher securities. Thus, the signature size is usually the primary source of overhead while using the proposed protocol. Regarding the public key sizes, plenty of schemes have the size at its minimum — the size of the hash function output. These two sizes are determined by the signature scheme selection and its parametrisation. However, the final overhead regarding the public key size is determined by the two important protocol parameters — `KEY_RATIOS` and `PRE_CERT`. These two parameters are described in the section 2.2.2 and section 2.2.2, respectively.

Instance name	PK size	SK size	Sig. size	Bit sec.	Key charges
HORST-RIPEMD-128	16	1048576	2176	64	32
HORST-RIPEMD-160	20	1310720	3400	80	25
HORST-SHA3-224	28	1835008	6664	112	18
HORST-SHA3-256	32	2097152	8704	128	16
HORST-SHA3-384	48	3145728	19584	192	10
HORST-SHA3-512	64	4194304	34816	256	8

Table 4.2 Byte sizes of a public key, a secret key, and a signature. Also, it displays the classical bit security (post-quantum security is half of the classical one) for recommended HORST signature scheme parametrisations (i.e. scheme instances) and the maximum number of signatures before the security drops below the target security. The provided values ignore the scheme vulnerability against 'weak message' attacks and against adaptive chosen message attacks. The reasons for this are discussed in the section 1.3.2.

4.2 Parametrising the re-authentication delay

Whenever a receiver misses some messages, be it due to network issues or due to intentional disconnect from the data stream, the protocol must be able to re-authenticate already-known identities within some reasonable time. The number of messages the receiver needs to receive from the sender with an already-known identity before labelling the messages as authenticated again is called 're-authentication time'. The time to re-authenticate also depends on what public keys the receiver has; the key concept is that the receiver will have some of the longer-living keys cached. Once the sender signs with one of these keys, the receiver can re-authenticate it again; this means the receiver must receive a certain number of messages to acquire keys from certain layers. Since the key for signing a message is sampled from the probability distribution, each layer has a probability p to be selected. Given that, to compute, for example, how many messages one has to receive to have received the key from the given layer with the probability at least q is:

$$\frac{\log q}{\log(1-p)}$$

In an ideal world, re-authentication time would always be possible after receiving just a single message. Even though this is possible, the data overhead would be huge, and, generally, this is not a practical direction to take. In order to keep overhead in moderation, one would need to sacrifice some of this re-authentication time. Generally, the bigger the message loss, the longer the re-authentication time, but one can select the `KEY_RATIOS` parameter to fit the distribution of message misses in the given use case.

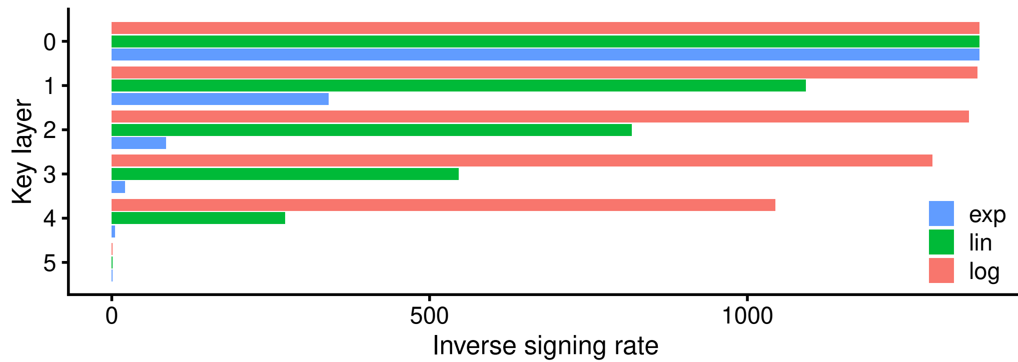
These include *exponential*, which is suitable for small message losses, *linear* uniformly distributes the possible misses throughout the re-authentication interval, and *logarithmic* is designed for greater message misses. The specific ratio definitions can be seen in the figure 4.1.

```
# Exponential
[1024, 256, 64, 16, 4, 1];
# Linear
[1354, 1083, 812, 542, 271, 1]
# Logarithmic
[1360, 1357, 1344, 1286, 1040, 1]
```

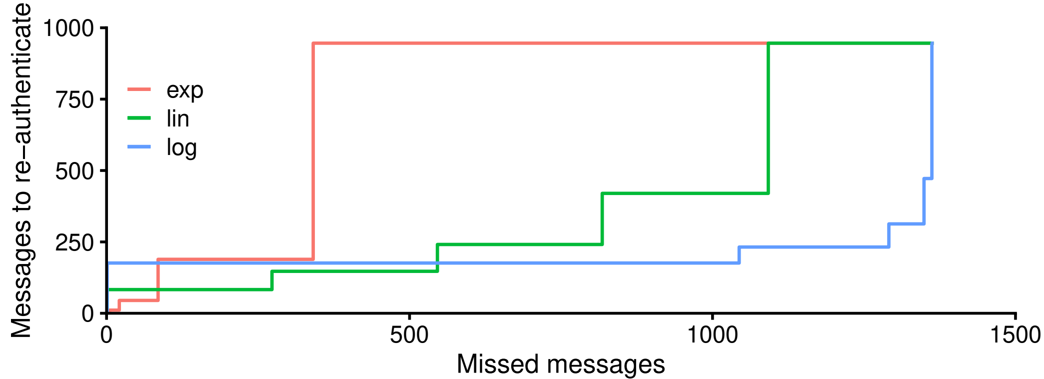
Figure 4.1 The suggested key ratios demonstrating the three different potential use cases. These are named exponential (`exp`), linear (`lin`), and logarithmic rates (`log`).

It is important to note that all these three ratio parameters yield the configuration with the same expected inverse sign ratio of the longest-living key layers. The inverse signing ratio x states that the key on the given layer will sign, on average, every x-th message; this determines for how long the key on each layer will be alive. The expected inverse signing ratios for the three parameters are shown in the figure 4.2a. We see that the longest-living key will sign roughly once per 1300 messages, or in other words, that the epoch of the layer 0 key is 1300. The figure 4.2b, the theoretical approximations of medians of re-authentication times are presented. This canonical variant represents the situation where each key can sign only a single message, and we are not sending any public keys in advance.

Although this may not seem practical, this can be used to calculate the approximation for the real parameters. One just multiplies the missed messages by the $(\mathrm{PC}+1) \cdot \mathrm{KEY_CHARGES}$ and approximation on how the specific parametrisation of the protocol will behave in terms of re-authentication time. To verify this statement, the practical measurements for $\mathrm{PC} = 1, \mathrm{KEY_CHARGES} = 1$ were conducted; the results are presented in the figure 4.3. Importantly, the x-axis values are doubled since the real-world implementation cannot work without pre-certification of public keys; thus, this is the minimal, practically measurable configuration. The visualisations show the median along with quartiles for both the theoretical approximation and the practically measured result. We see that the approximation is not very tight, but that is due to the fact that the pre-certification window is just one key, and every key can sign just once; as it will be later shown, once these parameters are raised to some reasonable number, the approximation work quite well.



(a) Expected inverse key signing rates of each layer for exponential, linear and logarithmic key ratios. The inverse signing rate x tells that the key from the given layer will, on average, sign every x-th message.



(b) Median approximation of the number of messages needed to re-authenticate after the given number of messages have been missed. This shows the base re-authentication interval — it means that this corresponds to the situation with only one key signature per key and without announcing the future keys. Increasing the `PC` and `KEY_CHARGES` parameters scales the x-axis values by a factor of $(\mathrm{PC}+1) * \mathrm{KEY_CHARGES}$. The results for realistic parameters are displayed in the figure 4.5

Figure 4.2

4.2.1 Measuring methodology

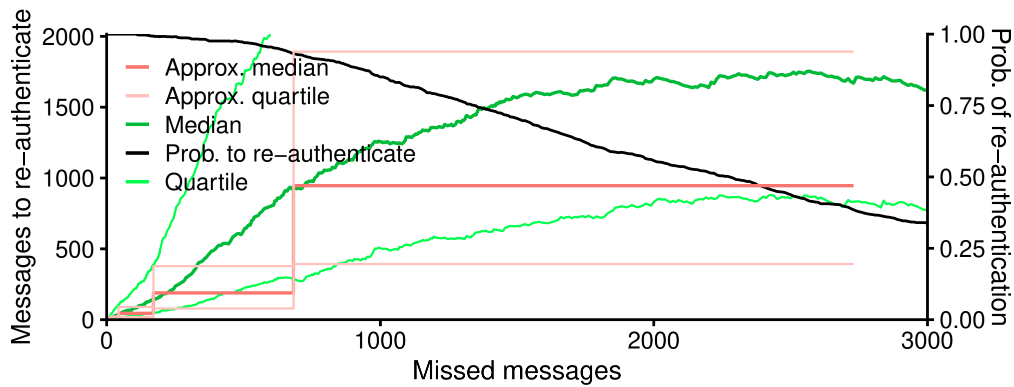
The practical measurements use the proof-of-concept implementation introduced in the chapter 3. To reduce the runtime, the simulated versions of the `Sender` and `Receiver` structures are used — `SenderSim` and ReceiverSim. The logic behind it is the same, but all the overhead irrelevant to the re-authentication is removed; such as signatures — they are replaced by just integers signalling what key signed the message and what keys are attached. The code to generate the re-authentication results presented is in the structure `Benchmarker` inside the directory `benchmark` in the HAB crate; it is a sub-project with its own README file. One should head there to reproduce the results, given that the seed is unchanged.

The re-authentication time was measured like this: The new sender and receiver are created with the required parameters. The receiver receives a sufficient number of messages to exceed the probability of 0.99 for having the layer 0 key. Then the given number of messages are not received by the receiver. And finally, the receiver starts to get the signed message once again. From that moment, the messages are counted; once the message is tagged as authenticated, the number is written to the results. Because it also can happen that the receiver may be unable to re-authenticate because it has lost too many messages, we do not iterate indefinitely. The limit is again the number of messages that contain a message signed by the longest-living key with the probability of 0.99. The measurement was done in thousand repetitions; in each, the new sender and receiver instances are instantiated.

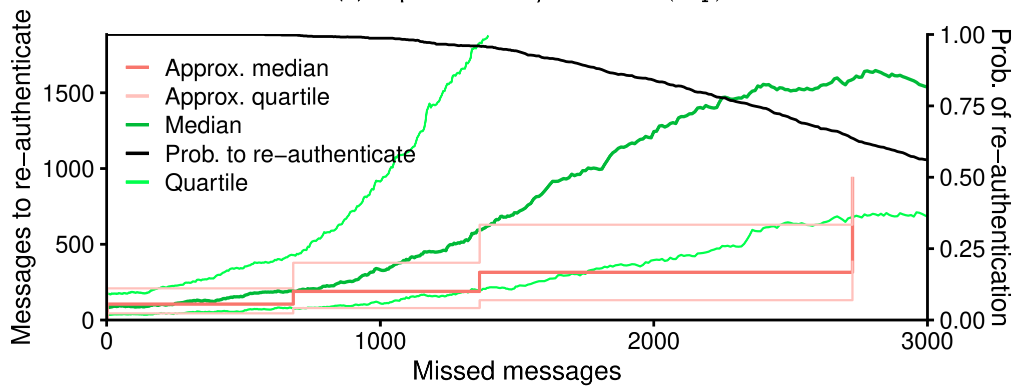
4.3 Data overhead trade-offs

When designing the parameters for the proposed protocol, one has to accept certain trade-offs. The most important one is the trade-off between the security of the authentication and the size of data that is transferred in addition to the actual payload — data overhead. The used signature scheme and its parameters determine the size of the signature as well as the size of the public key. The signature size is a significant portion of the overhead. The second part is attached public keys. Their number determines how large message misses can be and how fast the receivers are able to re-authenticate known identities.

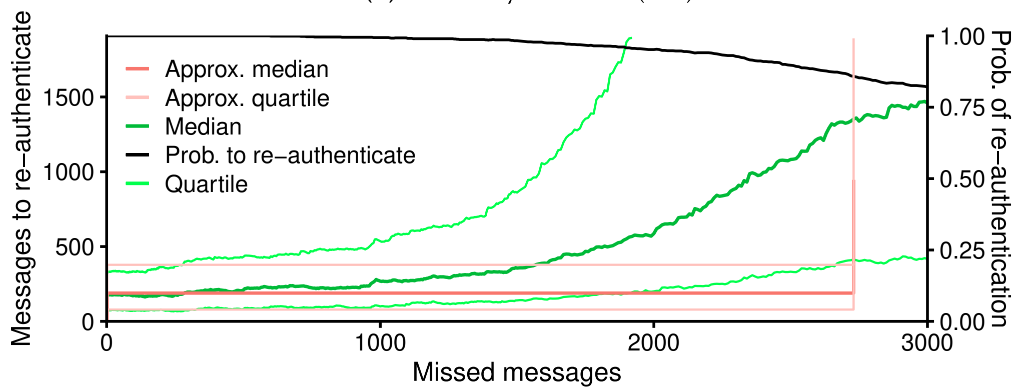
In the figure 4.4, the growing overhead of pieces is visualised for the previously presented HORST signature scheme instances (see the table 4.2). The colours correspond to the selected instances, and on the x-axis, we see the growing number of public keys attached to each message. The overhead grows faster for the instances with higher security due to bigger public keys, and it also starts



(a) Exponential key durations (`exp`)



(b) Linear key durations (`lin`)



(c) Logaritmic key durations (`log`)

Figure 4.3 Re-authentication times for the three suggested key ratios with minimal pre-certification parameter and single key charge (PC = 1, KEY_CHARGES = 1). Both the approximation and empirical results are shown; this sketches the basic behaviour of these key ratios. Increasing the `PC` and `KEY_CHARGES` parameters scales the x values by a factor of $(\text{PC} + 1) \cdot \text{KEY_CHARGES}$. The presented results are for PC = 1, and thus the base x values are scaled by a factor of PC + 1 = 2. The unscaled re-authentication intervals are displayed in the figure 4.2

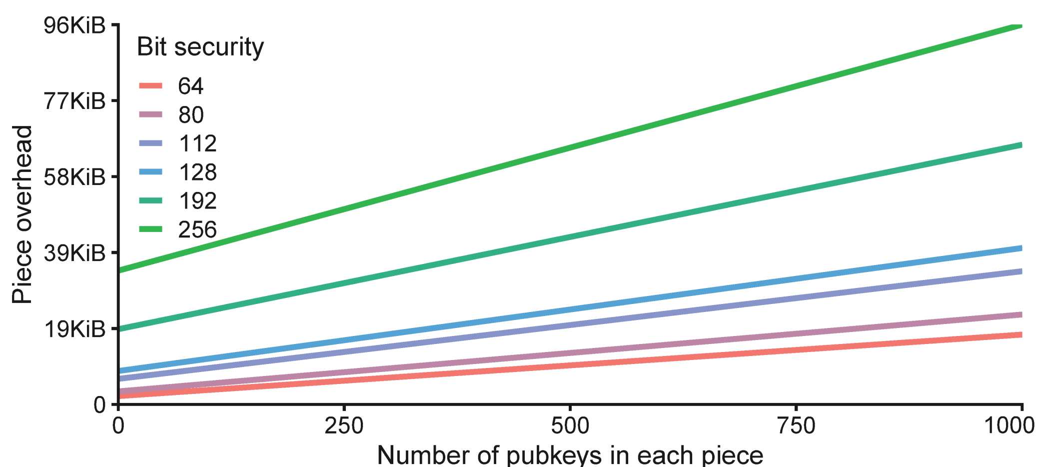


Figure 4.4 The trade-off between security, the number of attached public keys in each message and data overhead. The presented bit securities correspond to the suggested signature scheme instances. The growing number of attached public keys increases the message miss tolerance but at the same time increases the overhead. The higher the security level, the higher the signature and public key sizes — the growing public key sizes due to higher security cause the overhead grows faster for instances with higher securities..

significantly higher due to the signature size.

The importance of this overhead is specific to each use case. Some use cases may transfer large messages where overhead in tens of thousands of kibibytes is acceptable. In the case of large messages, one also has to consider the potential unreliability of the underlying network interface — as in our proof-of-concept implementation, for example. If one has a large message split into small UDP datagrams (e.g. of size 1500B), the probability of some datagram being lost grows. Receivers may then end up with an almost complete message missing just one piece, and the overall overhead may be even higher than if some reliable protocol is used to transfer the pieces or fragments. On the other hand, using some reliable protocol would add more latency to the delivery, especially if the data is distributed via multiple distributor nodes. This may not be acceptable for some real-time use cases.

Ultimately, one has to carefully weigh these trade-offs and design the parameters for the concrete use case. The used network interface also plays a role in total data overhead.

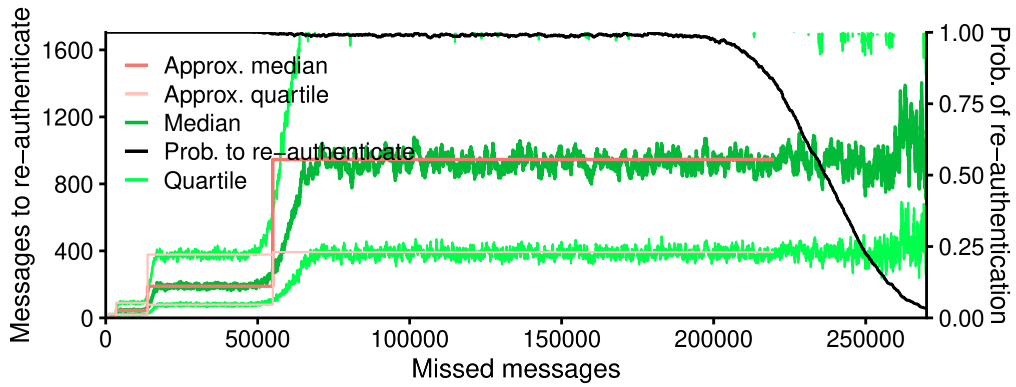
Fragment overhead Fragmenting data into smaller pieces generates additional data overhead, and it should be considered when designing the parameters. The format of fragments is shown in the figure 2.5.

4.4 Protocol behaviour with practical parameters

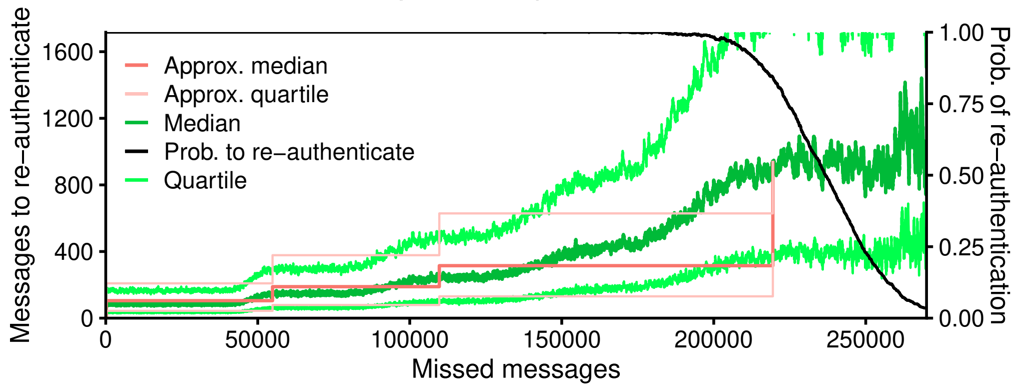
We did empirical measurements to validate that the protocol works with practical parameters, where it can withstand significant message losses. Also, to check that the approximation based on geometric distribution can be used for choosing the parameters and that is corresponds to the real implementation. The three key distribution parameters defined in the figure 4.1 were used with KEY_CHARGES = 20 and PRE_CERT = 8. That means that each key can generate twenty signatures, and the sender pre-certifies eight keys before using them for signing. The results are presented in the figure 4.5. This measured configuration is well usable in real-world scenarios but offers relatively small allowed packet losses; this parametrisation was selected for measurements to take reasonable time with enough repetitions. The configuration can be, of course, arbitrarily scaled by adding new layers and increasing the `PRE_CERT` parameter.

This configuration offers a decent message loss — up to two hundred thousand messages can be missed without losing the ability to re-authenticate. The time to re-authenticate depends on the missed number and the key distribution strategy used. One can see that in the case of practical protocols, the empirical results alternate around the theoretic approximations, which hints that the theoretical approximation can be used for the parameter design. Using this configuration, if the sender is broadcasting non-stop, sending one message each second, the receivers could miss still go offline for more than two days and will be able to re-authenticate. The are, of course, use cases where much greater message misses are usual; in cases like these, one can just add a few more layers to fit the desired re-authentication interval.

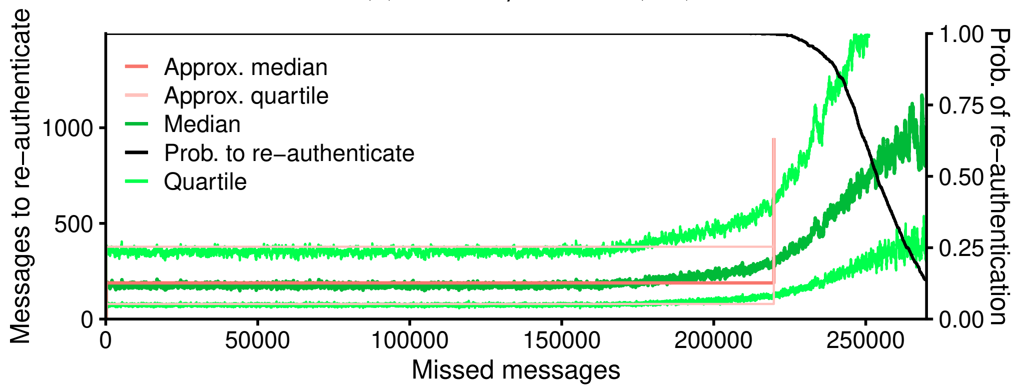
The detail of the behaviour of these configurations is shown in the figure 4.6; this is to see how the different configurations behave with smaller numbers of missed messages. We see that all the configurations can safely cover message misses caused by usual network losses. Also, it is visible that the exponential key selection guarantees fast re-authentications for shorter misses compared to the linear or logarithmic selection. In contrast, the logarithmic selection can re-authenticate much sooner if the number of misses grows significantly. The linear selection is a compromise between the two.



(a) Exponential key durations (`exp`)

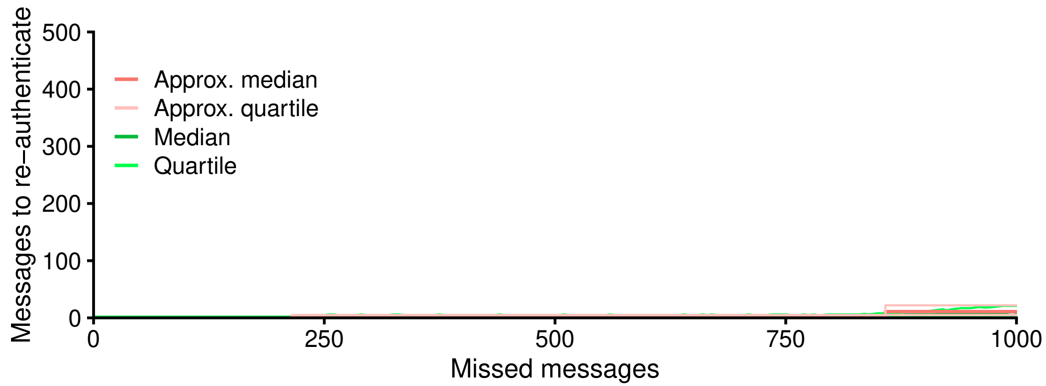


(b) Linear key durations (`lin`)

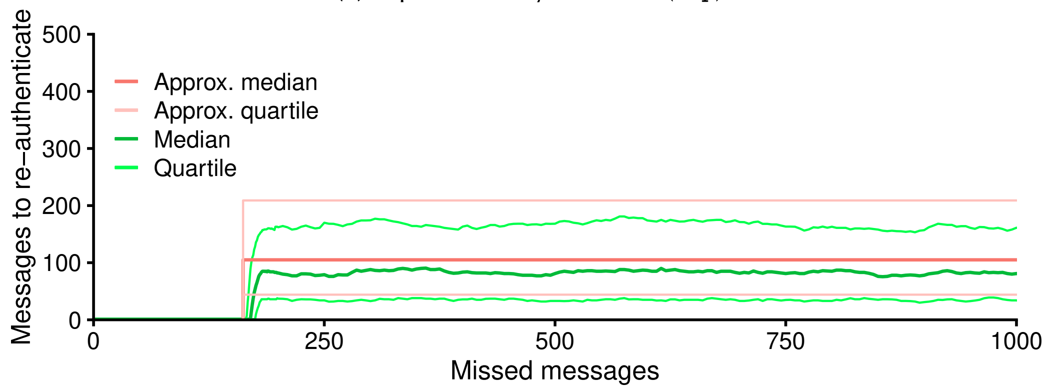


(c) Logaritmic key durations (`log`)

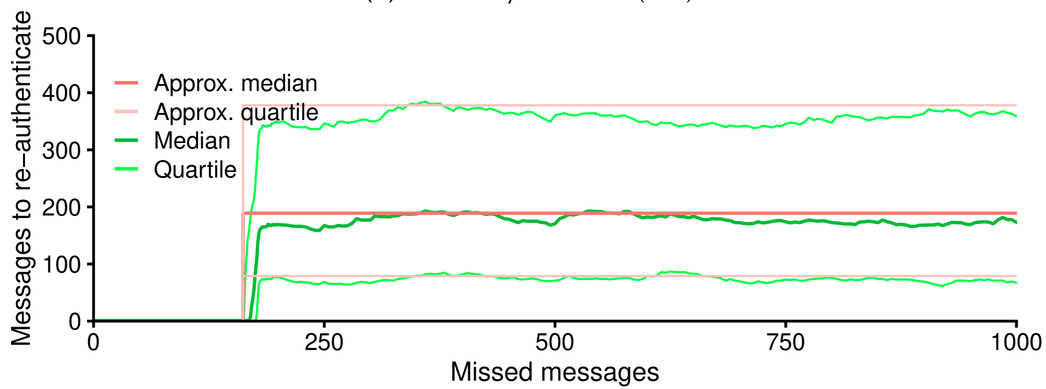
Figure 4.5 A measured time to re-authenticate after a specific number of messages is missed for practical parameters (KEY_CHARGES = 20, PRE_CERT = 8); this corresponds with the theoretical approximation figure 4.5. The three suggested key selection distributions are visualised.



(a) Exponential key durations (`exp`)



(b) Linear key durations (`lin`)



(c) Logaritmic key durations (`log`)

Figure 4.6 A measured time to re-authenticate after a specific number of messages is missed for practical parameters but showing only the small number of missed messages; this is the detail of the figure 4.5.

Conclusion

In conclusion, the thesis has presented a protocol for authenticated data streaming based on hash-based few-time signature schemes using petname identities (described in the chapter 2). The primary objective of this thesis was to evaluate the suitability of hash-based signatures for the authentication of data streams, with a particular emphasis on real-time applications (section 4.1). The proposed protocol is well-suited for use on the Internet, providing post-quantum security, simple implementation, and adaptability to a wide range of applications. It can serve as a quantum-robust alternative to current digital signatures and other techniques for data stream authentication. Notably, the protocol is compatible with any quantum-robust few-time signature scheme, offering data authentication and efficient identity management of known senders.

Moreover, the protocol design considers the usage of different network interfaces for sending messages. Therefore additional properties like (partial) reliability or confidentiality can be used without altering the protocol design (section 2.1.3). This protocol addresses the growing need for secure data streaming and broadcasting protocols that can withstand future threats, including those posed by advances in quantum computing.

While the protocol is highly adjustable and offers significant benefits, a potential drawback is the larger data overhead, particularly when there is insufficient data to be sent along with the signature (section 2.3, section 4.3). As such, additional parameter space examination is needed to reduce further the overhead caused by public keys attached to each piece. This thesis provides an initial exploration of practical configurations and demonstrates the impact of various parameters on protocol security, overhead, and the expected number of messages that can be lost without losing the ability to re-authenticate known identities (section 4.4). Despite measuring the specific parametrisation, the protocol can be scaled to withstand packet loss of arbitrary size. Additionally, developing alternative signature schemes with satisfactory runtime performance, small signature sizes, and further optimising the protocol's configuration will enable the protocol to be fine-tuned for specific use cases and requirements.

In summary, this thesis has enriched the field of authenticated data streaming

by proposing a post-quantum protocol that meets the proposed design criteria and is highly configurable. The proof-of-concept implementation of the protocol library with practical configurations and the real-world application for live audio streaming have demonstrated the viability of the proposed solution (section 3.1, section 3.2). As the demand for secure data streaming and broadcasting continues to grow, the work presented in this thesis provides a solid foundation for future advancements in the field.

Future work Specific directions on what can be done are stated to improve the protocol further.

- The security level of keys on each layer can vary — the longer-living keys need to have higher security, but the keys that are used more frequently can potentially use lower-security signatures; this can reduce the overall overhead since the signatures contribute significantly to it.
- Focus on reducing the number of public keys in each message to the minimum so receivers can still authenticate the stream of data without any authentication misses. The two strategies may not be optimal.
- A way the receivers could find the best data source for them would improve the overall usefulness of the protocol; for example, a distributed hash table with potential sources.
- Multiple senders can share an identity without sharing their secret keys; this can be used to form 'streaming communities'. The protocol can be extended to explicitly support this without the need for external synchronisation of senders.

Bibliography

[1] Roger Pantos and William May. *HTTP Live Streaming*. RFC 8216. Aug. 2017. DOI: 10.17487/RFC8216. URL: https://www.rfc-editor.org/info/rfc8216.

[2] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: 10.17487/RFC9147. URL: https://www.rfc-editor.org/info/rfc9147.

[3] Corporate Nist. "The digital signature standard". In: *Communications of the ACM* 35.7 (1992), pp. 36–40.

[4] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[5] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1 (2001), pp. 36–63.

[6] Adrian Perrig et al. *Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction*. RFC 4082. June 2005. DOI: 10.17487/RFC4082. URL: https://www.rfc-editor.org/info/rfc4082.

[7] Adrian Perrig. "The BiBa one-time signature and broadcast authentication protocol". In: *Proceedings of the 8th ACM Conference on Computer and Communications Security*. 2001, pp. 28–37.

[8] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/s0097539795293172. URL: https://doi.org/10.1137%2Fs0097539795293172.

[9] M. Ajtai. “Generating Hard Instances of Lattice Problems (Extended Abstract)”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 99–108. ISBN: 0897917855. DOI: 10.1145/237814.237838. URL: https://doi.org/10.1145/237814.237838.

[10] Jintai Ding and Albrecht Petzoldt. “Current state of multivariate cryptography”. In: *IEEE Security & Privacy* 15.4 (2017), pp. 28–36.

[11] Raphael Overbeck and Nicolas Sendrier. “Code-based cryptography”. In: *Post-quantum cryptography* (2009), pp. 95–145.

[12] Luca De Feo, David Jao, and Jérôme Plût. “Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies”. In: *Journal of Mathematical Cryptology* 8.3 (2014), pp. 209–247. DOI: doi:10.1515/jmc-2012-0015. URL: https://doi.org/10.1515/jmc-2012-0015.

[13] Arthur Evans, William Kantrowitz, and Edwin Weiss. “A User Authentication Scheme Not Requiring Secrecy in the Computer”. In: *Commun. ACM* 17.8 (1974), pp. 437–442. ISSN: 0001-0782. DOI: 10.1145/361082.361087. URL: https://doi.org/10.1145/361082.361087.

[14] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[15] Bruce Schneier. “Key Length”. In: *Applied Cryptography, Second Edition*. John Wiley & Sons, Ltd, 2015. Chap. 7, pp. 151–168. ISBN: 9781119183471. DOI: https://doi.org/10.1002/9781119183471.ch7. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119183471.ch7. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119183471.ch7.

[16] Ilya Mironov et al. “Hash functions: Theory, attacks, and applications”. In: *Microsoft Research, Silicon Valley Campus* (2005), pp. 1–22.

[17] Daniel J Bernstein. “Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete”. In: *SHARCS* 9 (2009), p. 105.

[18] Lov K Grover. “A fast quantum mechanical algorithm for database search”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.

[19] Gilles Brassard, Peter Høyer, and Alain Tapp. “Quantum cryptanalysis of hash and claw-free functions”. In: *LATIN’98: Theoretical Informatics: Third Latin American Symposium Campinas, Brazil, April 20–24, 1998 Proceedings 3*. Springer. 1998, pp. 163–169.

[20] W. Diffie and M. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[21] Michael O. Rabin. “DIGITALIZED SIGNATURES AND PUBLIC-KEY FUNCTIONS AS INTRACTABLE AS FACTORIZATION”. In: 1979.

[22] Leslie Lamport. *Constructing Digital Signatures from a One Way Function*. Tech. rep. CSL-98. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010. 1979. URL: https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/.

[23] Daniel J Bernstein et al. “The SPHINCS+ signature framework”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 2129–2146.

[24] Ralph C. Merkle. “A Certified Digital Signature”. In: *Annual International Cryptology Conference*. 1989.

[25] Chris Dods, Nigel P Smart, and Martijn Stam. “Hash based digital signature schemes”. In: *Cryptography and Coding: 10th IMA International Conference, Cirencester, UK, December 19-21, 2005. Proceedings 10*. Springer. 2005, pp. 96–115.

[26] Ralph Charles Merkle. *Secrecy, authentication, and public key systems*. Stanford university, 1979.

[27] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. “XMSS-a practical forward secure signature scheme based on minimal security assumptions”. In: *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*. Springer. 2011, pp. 117–129.

[28] Andreas Huelsing et al. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. May 2018. DOI: 10.17487/RFC8391. URL: https://www.rfc-editor.org/info/rfc8391.

[29] David A Cooper et al. “Recommendation for stateful hash-based signature schemes”. In: *NIST Special Publication* 800 (2020), p. 208.

[30] Daniel J Bernstein et al. “SPHINCS: practical stateless hash-based signatures”. In: *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*. Springer. 2015, pp. 368–397.

[31] Leonid Reyzin and Natan Reyzin. “Better than BiBa: Short one-time signatures with fast signing and verifying”. In: *Information Security and Privacy: 7th Australasian Conference, ACISP 2002 Melbourne, Australia, July 3–5, 2002 Proceedings 7*. Springer. 2002, pp. 144–153.

[32] Jean-Philippe Aumasson and Guillaume Endignoux. “Clarifying the subset-resilience problem”. In: *Cryptology ePrint Archive* (2017).

[33] Jean-Philippe Aumasson and Guillaume Endignoux. “Improving stateless hash-based signatures”. In: *Topics in Cryptology–CT-RSA 2018: The Cryptographers’ Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*. Springer. 2018, pp. 219–242.

[34] Jaeheung Lee et al. “HORSIC: An efficient one-time signature scheme for wireless sensor networks”. In: *Information Processing Letters* 112.20 (2012), pp. 783–787.

[35] Andreas Hülsing. “W-OTS+–shorter signatures for hash-based signature schemes”. In: *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*. Springer. 2013, pp. 173–188.

[36] Marc Stiegler. “Petname systems”. In: *HP Laboratories, Mobile and Media Systems Laboratory, Palo Alto, Tech. Rep. HPL-2005-148* (2005).

[37] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[38] Neal H Walfield and Werner Koch. “TOFU for OpenPGP”. In: *Proceedings of the 9th European Workshop on System Security*. 2016, pp. 1–6.

[39] Robert Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[40] Kjeld Borch Egevang and Paul Francis. *The IP Network Address Translator (NAT)*. RFC 1631. May 1994. DOI: 10.17487/RFC1631. URL: https://www.rfc-editor.org/info/rfc1631.

Appendix A

Structure of the attached software

The thesis attachment is an archive that contains the provided proof-of-concept library implementation and the example application for live audio broadcasting that uses the library. The structure of the attachment is displayed in the figure A.1. The `hab` directory contains the repository [1] of the HAB library, and the `audibro` directory is the repository [2] of the example application.

A.1 `hab`

`benchmark` The application which is using the library that was used for benchmarking the library.

`src` Rust source files for the library.

`Cargo.yml` A standard config file for Rust projects specifying the package metadata and used dependencies.

`README.md` A file with the basic information on how to use the library.

`LICENSE.txt` A file with the license of the project.

`.gitlab-ci.yml` A script file defining the behaviour in the GitLab pipeline.

A.2 `audibro`

`data` A directory with bundled-in audio files that can be streamed with the application.

`docs` A directory with documentation source files. Also, the generated documentation is outputted here.

[1] `https://gitlab.mff.cuni.cz/mejzlikf/hab`

[2] `https://gitlab.mff.cuni.cz/mejzlikf/audibro`

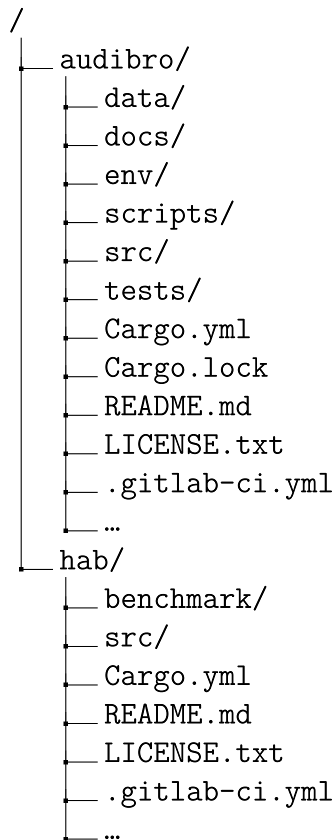


Figure A.1 The structure of the attached software archive displaying the important directories and files.

env A directory where the temporary files for senders and receivers are stored. For example, their identity files and identity graphs.

scripts A directory with convenience scripts that can be used to configure and launch the application.

src Rust source files for the library.

Cargo.yml A standard config file for Rust projects specifying the package metadata and used dependencies.

Cargo.lock A standard file for Rust projects locking in the specific dependency versions.

README.md A file with the basic information on how to use the library.

LICENSE.txt A file the license of the project.

.gitlab-ci.yml A script file defining the behaviour in the GitLab pipeline.

Appendix B

Using the HAB library

To use the library, one needs to include the hab library as a GIT dependency [1] in the Cargo.toml file in the target project. The source code is located in the public repository [2]. The library compilation will be handled by the cargo build tool for you. Naturally, you need a Rust compiler; the minimal supported version is 1.58 [3]. There are no additional direct dependencies required, but the library uses other Rust crates that may, on some platforms or in the future, require some libraries installed in the system. These are typically easy to resolve by using a system package manager.

With that, Sender and Receiver structs can be included in the source file. To configure those, the SenderParams and ReceiverParams need to be included; these will be used to configure the protocol. The configuration parameters are described in the section 2.3 and also in respective source files — sender.rs and receiver.rs. Both sender and receiver instances require some instance of a few-time signature scheme to be used. The library comes with the HORST signature scheme bundled in; it can be imported as HorstSigScheme. The scheme itself must be configured with corresponding parameters. To use different signature schemes, one can implement arbitrary schemes by implementing the FtsScheme trait. It can be used as the generic argument for the sender and receiver instances. The attached example of a live audio broadcast application is described in the next appendix C; it shows how the library can be integrated within a real-world application.

The source code itself and the developer documentation can be used to understand the internals in case of doing some extensions to the library. The developer documentation can be generated and viewed by running:

[1] https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html
[2] https://gitlab.mff.cuni.cz/mejzlikf/hab
[3] https://www.rust-lang.org/tools/install

```
cargo doc --open
```

B.1 Sender

Once the sender is instantiated and configured, it is rather simple to start broadcasting. The struct offers one method called `broadcast` that accepts one parameter; it is a reference to the byte buffer to be broadcasted to currently subscribed receivers. The function call is a blocking one and returns when data for all receivers have been dispatched via the network interface. The returned type is a result indicating an error if some occur.

B.2 Receiver

With the receiver instance configured and instantiated, the `receive` method is available to use. It takes no parameters and returns a result type of either `ReceivedMessage` or an error instance with the reason for failure. The `ReceivedMessage` structure consists of three fields — a message that was broadcasted, a verification status of the message and a sequence number of this message. The verification status is used to determine if the data are proven to be sent by the target identity, if someone certified by the target identity signed the message, or if the message is not verified at all.

Appendix C

Using the AudiBro application

The example application showcases the usage of the `hab` library in a simplified real-world scenario of live audio broadcasting software. The source code is located in the public repository [1]. To compile and run AudiBro, a Rust compiler is required with a version of at least 1.58 [2], and on Linux distributions, the ALSA library [3] is needed. Usually, it can be installed using the system's package manager; for example, for Debian, one can run the following:

```
sudo apt update
sudo apt install libasound2-dev
```

With all the dependencies in place, one can compile the program using the following command:

```
cargo build --release
```

Once the program is compiled, it can be run. The easiest way to try the example is to use prepared bash scripts to send correct arguments to the application. The pre-configured setup showcases three entities - one sender called Alice. Alice is the original sender that broadcasts the audio. The next actor is Bob, who receives data from Alice and at the same time works as a distributor. The third one is called Carol, who receives data from Bob. This simple setup showcases all three roles implemented by the protocol. It is recommended to have three terminal windows prepared and run each actor in one of them; it can be done with the following commands:

[1] `https://gitlab.mff.cuni.cz/mejzlikf/audibro`
[2] `https://www.rust-lang.org/tools/install`
[3] `https://wiki.debian.org/ALSA`

Figure C.1 The terminal UIs for the three pre-configured instances.

```
# An original sender Alice (broadcasting on the port 5000)
bash ./scripts/run-tui-sender-alice.sh
# A receiver and distributor Bob (broadcasting on the port 5001)
bash ./scripts/run-tui-receiver-bob-from-alice.sh
# A receiver Carol (receiving from Bob)
bash ./scripts/run-tui-receiver-carol-from-bob.sh
```

With all three instances running, one can use the terminal user interface at Alice's instance to live stream the audio. The showcase of the user interface is in the figure C.1. Three bundled audio files can be broadcasted or the default input device (if some are present). The user shall use up and down arrow keys to navigate to the option that is to be broadcasted and then use ENTER key to play it. The source of audio can be changed at any time. The receivers shall receive the audio data and play it to the user via the default audio output device. The receiver interface indicates whether the user listens to authenticated data in real-time.

The README file inside the repository contains additional information about options and configurations