HTWK Leipzig University of Applied Sciences

Hochschule für Technik, Wirtschaft und Kultur Leipzig
Faculty of Computer Science and Media

Master Thesis in Computer Science

Theoretical and Practical Aspects of the Migration to Post Quantum Cryptography

Author: Florian Schröck
Enrolment Number: 78147

April 19, 2024

First advisor: Prof. Dr. rer. nat. habil. Martin Grüttmüller
Second advisor: M. Sc. Stefan-Lukas Gazdag (genua GmbH)

Declaration

With this statement I declare, that I have independently completed the following master thesis. The thoughts taken directly or indirectly from external sources are properly marked as such.

This thesis was not previously submitted to another academic institution and has also not yet been published.

Florian Schröck Leipzig, April 19, 2024

Contents

1 Introduction **1**

2 Theoretical Background - Post Quantum Cryptography **5**

2.1 Code-based Cryptography 6

2.1.1 McEliece Cryptosystem 9

2.2 Lattice-based Cryptography 11

2.2.1 CRYSTALS-Dilithium 13

3 Post Quantum Cryptography Migration of GitLab - a Case Study **16**

3.1 Problem Statement 17

3.2 Related Work 21

3.2.1 Software Tools for Static Program Analysis 24

3.3 Chosen Approach 26

4 Implementation **29**

4.1 Cryptographic Inventory Compilation 29

4.1.1 Results 36

4.2 Migration Planning 38

4.3 Migration Execution 45

4.3.1 PQC Commit Signatures in git 45

4.3.2 Including Dilithium to `ssh_data` 46

5 Conclusion and Outlook **52**

6 References **54**

List of Tables **71**

List of Figures **72**

List of Source Code **73**

Acronyms **75**

Notation **76**

1. Introduction

Cryptography is an indispensable tool for modern civilization. Often unnoticed, it enables confidential communication over an insecure channel and the secure identification of communication partners. Online banking, e-commerce, digital administrative services, instant messaging - none of these would be trustworthy without the utilization of cryptography.

Yet there is an unprecedented challenge to traditional cryptographic algorithms' security: the emergence of Quantum Computers (QCs). All of the most popular cryptographic primitives currently in use deduce their security assumptions from the *expected* hardness to solve two mathematical problems on classical computers: the *factorization of natural numbers* and the *Discrete Logarithm Problem (DLP)*. The Rivest-Shamir-Adleman (RSA) encryption and signature schemes are the most prominent algorithms relying on the complexity of factorization. Digital Signature Algorithm (DSA), ElGamal encryption and signature, Diffie-Hellman (DH) as well as algorithms from Elliptic Curve Cryptography (ECC), on the other hand, are some of the most important representatives relying on DLP's hardness.

Cryptographic primitives like these facilitate services and protocols such as Domain Name System (DNS), Transport Layer Security (TLS), Secure Shell (SSH), Secure / Multipurpose Internet Mail Extensions (S/MIME) and X.509 certificates in Public Key Infrastructures (PKIs), to name just a few. They build the cornerstones of modern network services, such as the World Wide Web, and are essential for all of the real-world applications mentioned above.

However, all of this is endangered and very soon, all of the primitives must be replaced. QCs, leveraging the principles of quantum mechanics, will be able to break current cryptographic schemes with remarkable efficiency. Three algorithms, in particular, loom ominously over classical cryptography: Shor's algorithms for factorization and discrete logarithms ([Sho94]) solve both problems in quantum polynomial time. Additionally, for an arbitrary function executable on a QC, Grover's algorithm ([Gro96]) probabilistically finds the preimage of a given image. This can be used to mount Brute-Force-Attacks on symmetric encryption functions such as Advanced Encryption Standard (AES) or conduct a preimage search for cryptographic hash functions such as Secure Hash Algorithm (SHA) variants.

All three quantum algorithms are significantly faster than their fastest counterparts on classical computers, making previously intractable computational problems conveniently solvable. As regards factorization, the General Number Field Sieve ([LL93], [CP05, p. 278-301]) is currently considered the fastest classical, general purpose factorization algorithm for large numbers, being sub-exponential, but super-polynomial in the integer's number of digits (see [KY21, p. 366]). The same holds true for the DLP: though faster algorithms for special types of groups exist, Shank's Baby-Step-Giant-Step algorithm (see [Sha71]) is still considered the fastest algorithm for arbitrary groups. However, its runtime is exponential in the group order's number of digits (see [KY21, p. 372f.], [TR22,

p. 141]). While for preimage search there is no deterministic algorithm on a conventional computer which can beat Linear Search with $\mathcal{O}(n)$ attempts for domain size n, Grover's Algorithm has a high probability to successfully deliver a search result in only $\mathcal{O}(\sqrt{n})$ attempts.

All in all, QCs will lead to an exponential speed-up with respect to factorization and DLP and a quadratic speed-up regarding Brute-Force-Attacks on symmetric ciphers and hash functions. Yet the latter is clearly less worrisome: by doubling the length of symmetric keys or hash function's output, the same assumed security level is achieved compared to the time prior to the emergence of sufficiently powerful QCs. Bearing in mind this simple countermeasure, there is a general consensus that symmetric cryptography is less vulnerable to attacks by QCs than asymmetric cryptography ([Nat20, section "Background"], [Bun21, section 6.4, p. 62], [ANS22, p. 2]).

It is tempting to rely on the fact that QCs today are far away from being capable of breaking conventional cryptographic schemes. In [GE21, p. 13], the authors provide a time estimate of 8 hours to factor a 2048 bit RSA modulus on a QC with 23 million physical qubits. FIGURE 1 from the same source provides more detailed estimations in similar dimensions. Yet, the largest systems currently on the market provide fewer than 1200 qubits, such as Atom Computing's 1180 qubit system ([Com23]) or IBM's Condor with 1121 qubits ([Cor23c]). Furthermore, the sole number of qubits is not the only parameter that needs to be improved. Many more technical challenges still require massive amounts of resources and ingenuity to be solved. [Sic23b] provides a thorough analysis of the status of the various QC architectures under development.

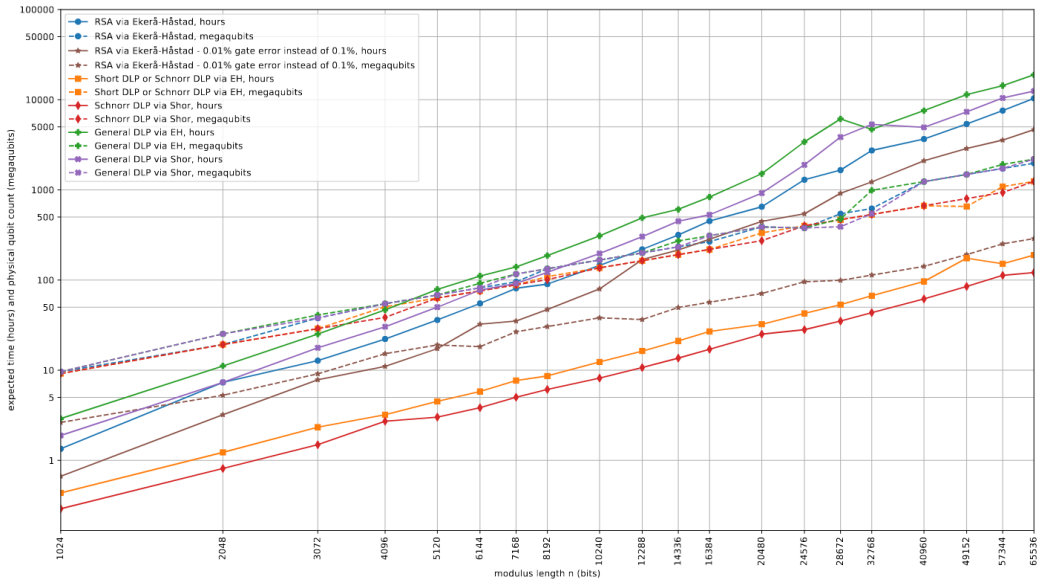

FIGURE 1. Estimation of time and qubits needed for factorization and solving of DLP using different approaches on QCs

Source: [GE21, Figure 1, p. 3]

The German Federal Office for Information Security (BSI) works under the conservative estimation that a cryptographically relevant QC will be available in the "early

2030s" ([Bun21, p. 35]). In [Sic23b], the authors expect it to take "at least one decade, more likely two" (p. 34) for QCs to execute Shor's algorithm for integers of relevant size. [MP23] contains more detailed estimations such as the one displayed in FIGURE 2, gained from a survey with 37 leading experts. Overall, there is little basis for an authoritative forecast predicting the quantum threat to materialize in the very near future.

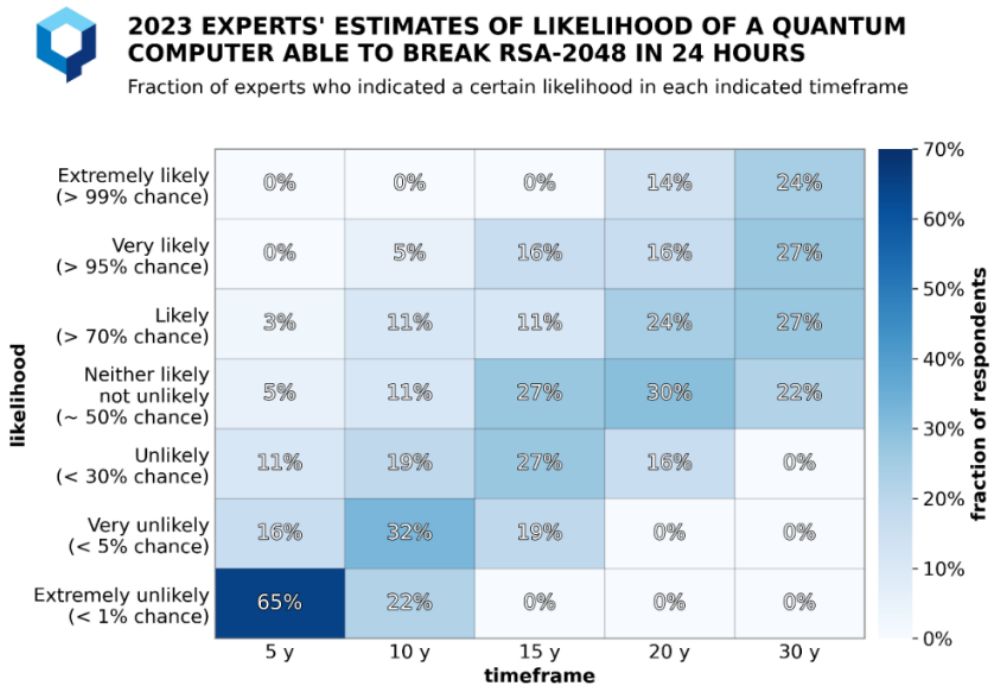

FIGURE 2. Experts' estimates of the likelihood of a QC breaking RSA-2048 in 24 hours

Source: [MP23, Figure 7, p. 21]

Nevertheless, the quantum threat is already a matter of concern today, not least for the following two reasons. Firstly, the state of development in quantum computing is progressing rapidly, driven by significant investments from governments, academia, and the private sector ([Sic23b, section 2.7], [Bun21, section 5]). Secondly, so called *Store-now-Decrypt-later-Attacks* must be taken into account: an adversary can save encrypted or signed data at one point, wait for the availability of a cryptographically relevant QC and use it to decrypt or forge the stored data at a later point in time. It is therefore no surprise that both NATO ([NAT24]) and the US Government ([Hou22]) have already passed strategies to prepare for the quantum threat.

Gladly, there is hope. To counter the threat posed by QCs, two techniques are evolving: Post Quantum Cryptography (PQC) and Quantum Key Distribution (QKD). While PQC tries to develop cryptographic primitives for conventional computers which withstand traditional and quantum attacks, QKD aims to utilize the quantum-physical possibilities of QCs themselves to provide secure methods of key exchange. However, the latter is still in a more experimental phase, and e.g. the German, French, Dutch and Swedish communication security authorities jointly released their view that QKD is not yet considered sufficiently mature. For the time being, they recommend relying on PQC (see [Bun+24]).

The history of PQC traces back to the early 2000s when experts began to concentrate research efforts on the development of alternative cryptographic schemes resilient to both traditional and quantum attacks. Various computational problems are currently considered highly likely to withstand the quantum threat. These problems have already been thoroughly studied, independently of their usability for cryptographic purposes. They predominantly stem from the mathematical branches of lattice theory, coding theory, multivariate algebraic equations or the isogeny theory of elliptic curves. In addition, hash-based cryptography explores the usage of digital signatures from cryptographic hash functions. The theoretical part of this thesis in section 2 briefly introduces the branches of PQC. The necessary mathematical background of code-based and lattice-based cryptography is presented in sections 2.1 and 2.2, together with one exemplary algorithm of each branch.

The development of cryptographic primitives alone is important, but insufficient: they have to be integrated into existing protocols and applications. This procedure is referred to as *PQC migration*. *Mosca's Theorem* ([Mos15]) reinforces the necessity for organizations to initiate their PQC migration as soon as possible by breaking down the quantum threat to a simple inequality: with $\boldsymbol{x}$ being the time which cryptographic keys need to remain secure, $\boldsymbol{y}$ the time necessary to finish a PQC migration and $\boldsymbol{z}$ the time until a QC can break a classical cryptographic algorithm in use, $\mathbf{x} + \mathbf{y} > \mathbf{z}$ would signify a "serious problem" with regard to Store-now-Decrypt-later-Attacks. Due to that, the practical part of this thesis aims to conduct a partial PQC migration for GitLab, a major source code management platform. Section 3 contains the problem statement definition (section 3.1) and a review of relevant advice from the PQC migration literature (section 3.2), before the chosen approach is set out (section 3.3).

The realization of this approach is described in section 4. Section 4.1 delineates the development of a software tool supporting the compilation of utilized cryptography, as well as the results which its application on GitLab produced. These results are the input for the migration planning in section 4.2, which finally led to the migration of two cryptographic functionalities explained in section 4.3. The thesis concludes with an outlook in section 5.

2. Theoretical Background - Post Quantum Cryptography

The aim of PQC is to provide cryptographic primitives comparable to those of classical cryptography while retaining the current level of security, even after the advent of sufficiently powerful QCs. As described in section 1, this predominantly regards asymmetric cryptography. In this respect, PQC research focuses on the development of two types of primitives: Key Encapsulation Mechanisms (KEMs) and digital signatures.

To use symmetric cryptography, e.g. in symmetric ciphers or Message Authentication Codes (MACs), entities have to negotiate a common secret. With their groundbreaking work [DH76], Diffie and Hellman initiated Public Key Cryptography in the 1970s and enabled a secure key exchange over an insecure channel with the Key Exchange Mechanism (KEX) named after them, which was adapted to work over elliptic curves as Elliptic Curve Diffie-Hellman (ECDH) in the 1980s ([Mil86], [Kob87]). Yet both algorithms will become unusable, as they rely on DLP's hardness. As an alternative to this prevalent method of secret negotiation, Cramer and Shoup formalized the idea of KEMs ([CS03]). While in a KEX both entities contribute to the shared secret, the main idea of a KEM is that one entity randomly generates a secret and uses asymmetric encryption to transfer it to the other entity (see [CS03, section 7]). The security of the KEM therefore depends on the security of the underlying asymmetric encryption, which now also has to withstand quantum attacks.

To achieve quantum security, PQC employs computational problems from various fields of mathematics and uses them to construct trapdoor functions for cryptographic primitives. Other than factorization and DLP, these problems are expected to be hard to solve for both conventional and quantum computers.

Code-based cryptography is based on the General Decoding Problem (GDP) from coding theory, in which one tries to decode a message that was encoded and deliberately pertubated with a random error. Further details are given in section 2.1.

Lattice-based cryptography's foundation is the theory of mathematical lattices, which are linear combinations of independent vectors with integer coefficients. Several problems exist for which even after decades of research there are no polynomial algorithms known. The most common problems consider the search for lattice vectors which are as short as possible or as close as possible to a given point. Section 2.2 elaborates on lattice-based cryptography.

For completeness isogeny-based, multivariate and hash-based cryptography shall be mentioned as further branches of PQC. Isogeny-based cryptography's security relies on the expected hardness of the isogeny problem, i.e. the search for a surjective morphisms inducing a group homomorphism between elliptic curves' point groups. The complexity of solving systems of multivariate polynomial equations over finite fields is the cornerstone of multivariate cryptography, while hash-based cryptography is not predominantly mathematically motivated, but uses the preimage resistance of cryptographic hash functions such as SHA-256 to construct digital signatures. An overview of all different branches is given in [BBD09] and [De 17].

2.1. **Code-based Cryptography.**

Code-based cryptography is based upon the mathematical subject of *Coding Theory*. In a broader sense, Coding Theory covers the study of mathematical tools to reach the following goals regarding the transmission of data ([see Big08, p. 3f.]):

1. **Economy**, i.e. reducing the amount of transferred data by *Data Compression*
2. **Reliability**, i.e. safeguarding the transmission of the correct information by *Error Recognition* or better *Error Correction*
3. **Security**, i.e. protecting the privacy of transmitted data by *Cryptography*

In a more narrow sense used in the course of this thesis, Coding Theory mainly deals with the second goal (some authors refer to this as *Channel Coding* in contrast to *Source Coding* covering data compression [see Rot06, p. 2f.]).

The basic idea is that the sender *encodes* data in a way that the receiver can (i) recognize transmission errors in the sent code word (e.g. to ask for a retransmission) or even better, correct them on his own and (ii) uniquely (and efficiently) *decode* the correct code word. The following section follows chapter 1 and 2 of [Rot06] with minor adaptations.

Definition 2.1 (Alphabet, Words, Coding Scheme)**.**

Let A be a finite set with $|A| \geq 2$ called **alphabet** and $k \leq n \in \mathbb{N}_0$. A^n contains **words** of length n with $A^0 := \{\varepsilon\}$ for the **empty word** ε. A **coding scheme** consists of an injective **encoding** $e : A^k \to A^n$ as well as a surjective **decoding** $d : A^n \to A^k$ so that d reverts c, i.e. $d \circ c = \mathrm{id}_{A^k}$.

A^k contains **messages** with **message length** k, $e(A^k)$ is called a **code**, its elements **code words** with **code length** n.

Some authors distinguish between separate input and output alphabets. This level of generality is not needed here: if not explicitly stated otherwise, A will be the set $[m] := \{0, \ldots, m\}$ for an $m \in \mathbb{N}_0$ and therefore identified with the ring $\mathbb{Z}_m := \mathbb{Z}/m\mathbb{Z}$. Due to that, one defines $\varepsilon = 0 := 0^n$ in A^n. In the even more restrictive case of *Linear Codes*, A needs to be a (finite) field. $|A|$ will be a prime power p^n, so $A = \mathbb{F}_{p^n}$ a finite extension of the Galois Field with p elements. The most relevant case is $A = \mathbb{Z}_2 = \mathbb{F}_2$.

Definition 2.2 (Linear Code)**.**

Let $\mathbb{F}$ be a finite field, $n \in \mathbb{N}$. A **linear code** C is a subspace of the vector space $\mathbb{F}^n$.

Basic Linear Algebra yields that every subspace can be written both as an image and a kernel of certain linear maps (and vice versa, images and kernels of linear maps are subspaces). This leads to the following definition.

Definition 2.3 (Generator Matrix, Parity Check Matrix)**.**

If there is a matrix $G \in \mathbb{F}^{k\times n}$ such that

$$C = \{xG = G^T x \mid x \in \mathbb{F}^k\} = \mathrm{im}_{\mathbb{F}}(G^T)$$

G is called a **generator matrix** for C. If a matrix $P \in \mathbb{F}^{(n-k)\times n}$ fulfils

$$C = \{Px = 0 \mid x \in \mathbb{F}^n\} = \ker(P)$$

P is called **parity check matrix** for C.

As the name suggest, generator matrices are useful to *generate* the set of all code words from the set of all messages, while parity check matrices are convenient to *check* if a word received after transmission is indeed a codeword. The distance between words is determined by their *Hamming Distance*.

Definition 2.4 (Hamming distance, Hamming weight)**.**

Let $A = \mathbb{Z}_m$ for $m \in \mathbb{N}$ be a finite alphabet.

$$\begin{aligned} d_H : A^n \times A^n &\to \mathbb{N}_0 \\ (x, y) &\mapsto |\{i \in [n] \mid x_i \neq y_i\}| \end{aligned}$$

is called **Hamming distance** on A^n. Furthermore

$$\begin{aligned} wt : A^n &\to \mathbb{N}_0 \\ x &\mapsto d_H(x, 0) = |\{i \in [n] \mid x_i \neq 0\} \end{aligned}$$

is called the **(Hamming) weight** (of x).

Indeed the Hamming distance is a metric, i.e. it is symmetric, positive definite and fulfils the triangle inequality ([see Rot06, p. 6]). In groups (such as $\mathbb{Z}_m$), there is a useful connection between the two concepts.

Corollary 2.5 (Relation between Hamming distance and weight)**.**

In any group A, $n \in \mathbb{N}$ and $x, y \in A^n$ holds $d_H(x, y) = wt(x - y)$

Proof. $d_H(x, y) = |\{i \in [n] \mid x_i \neq y_i\}| = |\{i \in [n] \mid x_i - y_i \neq 0\}| = wt(x - y)$ □

The higher the distance between code words, the less probable is the case that transmission errors alter a transmitted code word to another code word. This scenario is particularly bad, because the receiver can impossibly recognize that an error occurred. Therefore, the *Minimal Distance* of a code is of particular interest.

Definition 2.6 (Minimal distance, Minimal weight)**.**
Let C be a linear code. The **minimal distance of** C is defined as

$$d_H(C) := \min\{d_H(x, x') \mid x \neq x' \in C\}$$

The **minimal weight of** C is defined as

$$wt(C) := \min\{wt(x) \mid \varepsilon = 0 \neq x \in C\}$$

Indeed, both quantities are identical for linear codes (see [Wil99, Lemma 1.2.4, p. 15f.]).

Proposition 2.7 (Minimal Distance = Minimal Weight for linear codes)**.**
For all linear codes C of message length n over a field $\mathbb{F}$ holds $d_H(C) = wt(C)$.

Proof. The Hamming distance is invariant under translation:

$$d_H(x, y) = d_H(x + z, y + z)$$

holds for all $x, y, z \in \mathbb{F}^n$, because $x_i = y_i \Leftrightarrow x_i + z_i = y_i + z_i$. Due to that, one sees

$$\begin{aligned} d_H(C) &= \min\{d_H(x, y) \mid x \neq y \in C\} \\ &= \min\{d_H(x - y, y - y = 0) \mid x \neq y \in C\} \\ &= \min\{wt(x - y) \mid x \neq y \in C\} \\ &= wt(C) \end{aligned}$$

□

Definition 2.8 (Code Parameter)**.**

For a Code C of length n with $k = \dim(C)$ and $d = d_H(C)$ one refers to $[n, k, d]$ as the **parameters of** C. C is called C a $[n, k, d]$-code.

During decoding, the receiver tries to find the code word with minimal Hamming distance to the received word, i.e. the receiver tries to solve the following problem.

Problem 2.9 (General Decoding Problem (GDP))**.**

Let C be a linear $[n, k, d]$-code over a finite field $\mathbb{F}$ and $y \in \mathbb{F}^n$ (the received word).

Find a code word $x \in C$ minimising the Hamming distance $d_H(x, y)$ to y.

An equivalent statement of the GDP uses the *syndrome* of a word.

Definition 2.10 (Snydrome of a word)**.**
Let C be a linear $[n, k, d]$–Code with parity check matrix $P \in \mathbb{F}^{(n-k)\times n}$ over a finite field $\mathbb{F}$ and $y \in \mathbb{F}^n$. The **syndrome** of y is defined as $Py \in \mathbb{F}^{n-k}$.

The syndrome is used during *Syndrome Decoding*, which is equivalent to solving GDP, as the following proposition from [Ber15, p. 6f.] shows.

Proposition 2.11 (Equivalence of GDP and Syndrome Decoding)**.**

Let C be a linear $[n,k,d]$-Code with parity check matrix $P \in \mathbb{F}^{(n-k)\times n}$ over a finite field $\mathbb{F}$, $y \in \mathbb{F}^n$ and $s = Py$ the syndrome of y. The following statements are equivalent:

1. *$x \in C$ minimises the Hamming distance $d_H(x,y)$*
2. *$x = y - z$ for a z with minimal weight $wt(z)$ such that $Pz = s$*

Proof. With $x = y - z \in \mathbb{F}^n$, the following equivalencies hold:

1. $x \in C \Leftrightarrow Px = P(y-z) = Py - Pz = 0 \Leftrightarrow Pz = s$
2. x minimises the Hamming distance $d_H(x,y) \underbrace{\Leftrightarrow}_{2.5} wt(y-x) = wt(z)$ is minimal

□

The capacity of a code to detect and correct errors is related to its minimal distance. A $[n,k,d]$-code C is called $(d-1)$-*error-detecting* and $\lfloor \frac{d}{2} \rfloor$-*error-correcting*. These are upper limits: if d errors appear, one code word could be altered to another and the receiver cannot recognize it, as mentioned above. If more than $\lfloor \frac{d}{2} \rfloor$ error happen, the received word could be closer to another code word than the one sent by the sender, which would lead to a wrong decoding.

Since there are only finitely many code words, a naive way of decoding would be to enumerate all code words and look for the one closest to the received word. As long as not more than $\lfloor \frac{d}{2} \rfloor$ errors happen, this would guarantee a correct decoding for any $[n,k,d]$-code. As there are $|\mathbb{F}|^k$ code words, this algorithm would be exponential in k and therefore infeasible for large dimensions. In 1978, Berlekamp, McEliece, and van Tilborg showed that GDP is $\mathcal{NP}$-complete, using the Syndrome Decoding formulation (see [BMvT78]). It is therefore not surprising that so far no efficient classical or quantum algorithm is known to solve GDP. So generally speaking, the computationally complex part of a Coding Scheme is not the encoding of a message, but the decoding of a received word.

2.1.1. **McEliece Cryptosystem.**

Even if GDP's $\mathcal{NP}$-completeness reinforces the *general*, i.e. worst case, hardness assumption of transmission error removal, there are codes for which efficient error correction algorithms exist. In the beginning of the 1970s, the Soviet mathematician Goppa introduced such a class of algebraic codes later named after him (see [Gop70], [Gop71], [Gop72] and [Ber73] for a summary of the work in English).

Definition 2.12 (Binary Goppa Codes)**.**

Let $n, m \in \mathbb{N}$ and $g \in \mathbb{F}_{2^m}[X]$ be an irreducible polynomial of degree t. Let $L = (\alpha_1, \ldots, \alpha_n) \in \mathbb{F}_{2^m}^n$ with $\alpha_i \neq \alpha_j$ for $i \neq j$ such that $g(\alpha_i) \neq 0 \ \forall i \in [n]$ be the *support of* g.

The **Binary Goppa code** $\Gamma(L, g)$ is defined as

$$\Gamma(L, g) := \{c \in \mathbb{F}_2^n \mid \sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g\}$$

It can be shown that Goppa codes are linear by explicitly constructing parity check and generator matrices and that Goppa codes $\Gamma(L, g)$ with $deg(g) = t$ are $[n, k, d]$-codes with $k \geq n - mt$ and $d > 2t$ (see [EOS07, p. 157-159]). This implies in particular that they are t-error-correcting. Indeed, in 1975 Patterson published an efficient error correction (or decoding) algorithm for Goppa codes (see [Pat75]) that corrects up to t errors, if the generator matrix is known. This enabled McEliece to publish the following cryptosystem (see [McE78]).

Definition 2.13 (McEliece Cryptosystem)**.**

Key Generation

1. Select suitable $t < n \in \mathbb{N}$
2. Randomly select an irreducible polynomial $g \in \mathbb{F}_{2^m}[X]$ of degree t with support $L = (\alpha_i)_{1 \leq i \leq n}$
3. Construct the $k \times n$ generator matrix G of $\Gamma(L, g)$
4. Randomly select a $k \times k$ invertible matrix S and a $n \times n$ permutation matrix P
5. Calculate $G' = SGP$

Public Key: (G', t) *Private Key:* (G, S, P)

Encryption of $u \in \mathbb{F}_2^k$

1. Randomly select an error vector $z \in \mathbb{F}_2^n$ with $wt(z) = t$
2. Calculate $enc(u) = x = uG' + z$

Decryption of $x \in \mathbb{F}_2^n$

1. Calculate $x' = xP^{-1} = (uG' + z)P^{-1} = uSG + zP^{-1}$
2. Decode x' using Patterson's algorithm and receive $u' = uS$
3. Calculate $u'S^{-1} = u$

The decryption is correct: because P does only permute the entries of z, $wt(zP^{-1}) = wt(z) = t$ and Patterson's algorithm can correct up to t errors.

The security of the McEliece cryptosystem relies on two assumptions. Firstly, it is assumed that it is computationally infeasible to recover G from $G' = SGP$ (and optionally

additional information, depending on the chosen attack model) if n and t (which influences k) are chosen large enough. Otherwise, it would be possible for an adversary to obtain the private key. Secondly, if an adversary could efficiently solve the GDP, he could just decode x without knowing G. This is the point at which the cryptosystem relies on GDP's hardness.

So far it was not possible for attackers to structurally attack suitably parameterized McEliece cryptosystems (see [Ber+22a, section 3], [Sin20, section 6]). However, the cryptosystem has one frequently mentioned disadvantage. As its public and private keys contain large matrices, key sizes are large too (see [Sin20, p. 23]). Even if many attempts have been made to decrease key sizes while retaining the security level, most notably the one by Niederreiter ([Nie86], see [Liu+22] for an overview of the other), key sizes still remain an impediment for McEliece's popularity.

Nevertheless, an optimized McEliece cryptosystem was chosen as the asymmetric encryption of the Classic McEliece KEM ([Ber+23]). In the variant aiming to offer the highest security level, the public key needs 1357824 Byte = 1326 KiB ≈ 1,3 MiB and the private key 14120 Byte ≈ 13,8 KiB. Classic McEliece is one of three remaining submissions to the fourth round of the US National Institute for Standards and Technology (NIST) PQC Standardization process (see [Nat17]), which aims to standardize alternative candidates next to the ones already chosen after round three in 2022 (see section 3).

2.2. **Lattice-based Cryptography.**

Lattice-based cryptography utilizes certain problems considering mathematical lattices to construct cryptographic trap-door functions and goes back to the works of Ajtai ([Ajt96]) and Hoffstein, Pipher, and Silverman ([HPS98]). To give a brief overview, this section follows [Buc+17, section 1] and [Pau22, section 2.4.2].

Definition 2.14 (Lattice, Base, Matrix, Rank)**.**

Let $m \leq n \in \mathbb{N}$ and $b_1, \ldots, b_m \in \mathbb{R}^n$ be $\mathbb{R}$-linear independent vectors. The **n-dimensional lattice over** $b_1, \ldots, b_m$ is defined as

$$L(b_1, \ldots, b_m) := \langle b_1, \ldots, b_m \rangle_{\mathbb{Z}} = \{\sum_{i=1}^{m} x_i b_i \mid x_i \in \mathbb{Z}\}$$

The tuple $(b_1, \ldots, b_m)$ is called a **base** of $L(b_1, \ldots, b_m)$. If written as columns of a matrix $A \in \mathbb{R}^{n \times m}$, A is called a **matrix** of $L(b_1, \ldots, b_m)$. The **rank** of $L(b_1, \ldots, b_m)$ is defined as $rank(A)$.

Lattices have multiple bases or matrices, however two matrices M and M' span the same lattice L, iff there is transition matrix T which is unimodular, i.e. $M = TM'$ with $T \in \mathbb{Z}^{m \times m}$ and $|det(T)| = 1$. Therefore, the rank is well defined (see [Buc+17, p. 15]). Of particular interest are short lattices vectors. The concept of shortness is embedded into the following definition.

Definition 2.15 (Successive Minima of a Lattice)**.**

Let $L \subset \mathbb{R}^n$ be a lattice with $rank(L) = m$. For $i \in \{1, \ldots, m\}$ the **i-th minimum of L** is defined as the radius $r \in \mathbb{R}_{>0}$ of the smallest n-dimensional ball $U_n(0, r) := \{x \in \mathbb{R}^n \mid \|x\| \leq r\}$ that contains i linear independent lattice points:

$$\lambda_i(L) := \inf_{r \in \mathbb{R}_{>0}} \{\dim\langle L \cap U_n(0, r)\rangle_{\mathbb{Z}} \geq i\}$$

$\lambda_1(L) \leq \cdots \leq \lambda_m(L)$ are called the **successive minima** of L, in particular $\lambda_1(L)$ is the shortest length of a lattice point.

Lattices are thoroughly studied mathematical objects, yet for some computational problems relating to them there are no efficient algorithms known for neither classical nor quantum computers. The most important ones are defined below. For their definitions, let $m \leq n \in \mathbb{N}$ and $L \subset \mathbb{R}^n$ be a n-dimensional lattice of rank m with successive minima $\lambda_1 \leq \ldots \leq \lambda_m$.

Problem 2.16 (Shortest Vector Problem (Approximate Search Version) - SVP_γ)**.**

Given $\gamma \in \mathbb{R}_{\geq 1}$, find $v \in L \setminus \{0\}$ such that $\|v\| \leq \gamma\lambda_1$.

A special case of Problem 2.16 is SVP_1 which simply asks to find a lattice vector with minimal length. Most lattice problems can be formulated as search or decision problems. The decision version of SVP_γ is GapSVP_γ.

Problem 2.17 (Shortest Vector Problem (Approximate Decision Version) - GapSVP_γ)**.**

Given $\gamma \in \mathbb{R}_{\geq 1}$ and $d \in \mathbb{R}_{>0}$, decide if $\lambda_1 < d$ or $\lambda_1 \geq \gamma d$.

There are many more classical lattice problems that for example ask for particularly short bases such as the Shortest Independent Vectors Problem (SIVP) or vectors arbitrarily close to a fixed vector such as the Closest Vector Problem (CVP) (see [Buc+17, section 1.2] for a compilation).

The fastest known algorithm to solve SVP_γ either employ a lattice reduction with the LLL algorithm ([LLL82]) and / or the Block-Korkine-Zolotarev algorithm (BKZ, see [Sch87], [SE94]), both of which are exponential in the lattice dimension n (see [WXY23, p. 273]). Currently, there is also no quantum algorithm known to significantly speed up the search for shortest vectors (see [Alb+23]).

Yet the problems used to actually construct lattice-base cryptographic schemes are no lattice problems in the narrow sense, but intermediate problem to which lattice problems such as SVP_γ can be reduced. The existence of a reduction allows to translate an instance of a hard lattice problem to an instance of an intermediate problem and solve the latter to ultimately solve the lattice problem. As the lattice problem is expected to be hard, the existence of an efficient solution to the intermediate problem is unlikely. The most

important example is the Learning with Errors (LWE)-Problem, which requires to define a LWE-Distribution first.

Definition 2.18 (LWE Distribution $A_{s,\chi}$)**.**

Let $n \in \mathbb{N}$, $q \in \mathbb{Z} \setminus \{0\}$, χ be a distribution over $\mathbb{Z}_q$ and $s \in \mathbb{Z}_q^n$ a secret vector chosen uniformly at random. The **LWE-Distribution** $A_{s,\chi}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is generated by the following sampling algorithm:

1. Choose $a \in \mathbb{Z}_q^n$ uniformly at random
2. Choose $\varepsilon \in \mathbb{Z}_q$ according to the distribution χ
3. Compute $b = \langle a, s \rangle + \varepsilon \bmod q \in \mathbb{Z}_q$
4. Output $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$

Problem 2.19 (Learning with Errors - LWE)**.**

Let $m, n \in \mathbb{N}$, $q \in \mathbb{N}_{>0}$, $s \in \mathbb{Z}_q^n$ and χ be a distribution over $\mathbb{Z}_q$ and $A_{s,\chi}$ be a LWE-Distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$. Given m samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, solve the following problem:

- **Search Version sLWE:** Given the samples $(a_i, \langle a_i, s \rangle + \varepsilon_i)$ from $A_{s,\chi}$, find $s \in \mathbb{Z}_q^n$
- **Decision Version dLWE:** Decide with non-negligible probability, if m samples are either distributed according to a uniformly random distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ or $A_{s,\chi}$ for an $s \in \mathbb{Z}_q^n$

Regev proofed the equivalence of sLWE and dLWE and provided a reduction from GapSVP_γ to LWE ([Reg09]). This reduction has two remarkable properties: it is quantum and worst-case-to-average-case, i.e. if there exists an efficient algorithm solving a random LWE instance, then there exists an efficient quantum algorithm to solve any GapSVP_γ (i.e., it solves the worst case instance of GapSVP_γ).

To increase the computational efficiency of LWE-based schemes, Ring-LWE was proposed by Lyubashevsky, Peikert, and Regev as a modification to the original LWE (see [LPR10]). It replaces the ring $\mathbb{Z}_q$ with a quotient ring $R_q = \mathbb{Z}_q[X]/(f)$ with $f \in \mathbb{Z}_q[X]$. A common choice is $f = X^n + 1$ for a $n \in \mathbb{N}$. In the same source the authors provided a quantum reduction from SVP_γ to Ring-LWE. To go even further, Module-LWE replaces the ring R_q by the $\mathbb{Z}_q$-module R_q^d. Module-LWE can be reduced to Ring-LWE and vice versa ([AD17]). It's assumed hardness, based on the chain of reductions beginning at SVP_γ, is necessary for the digital signature CRYSTALS-Dilithium.

2.2.1. **CRYSTALS-Dilithium.**

CRYSTALS-Dilithium is one of the three digital signature algorithms chosen in the NIST PQC standardization process ([Nat17]). FIGURE 7 displays an overview of a

simplified version as provided by the authors. The differences to the submitted and implemented version mainly regard performance improvements and provisions to decrease the size of keys and signatures.

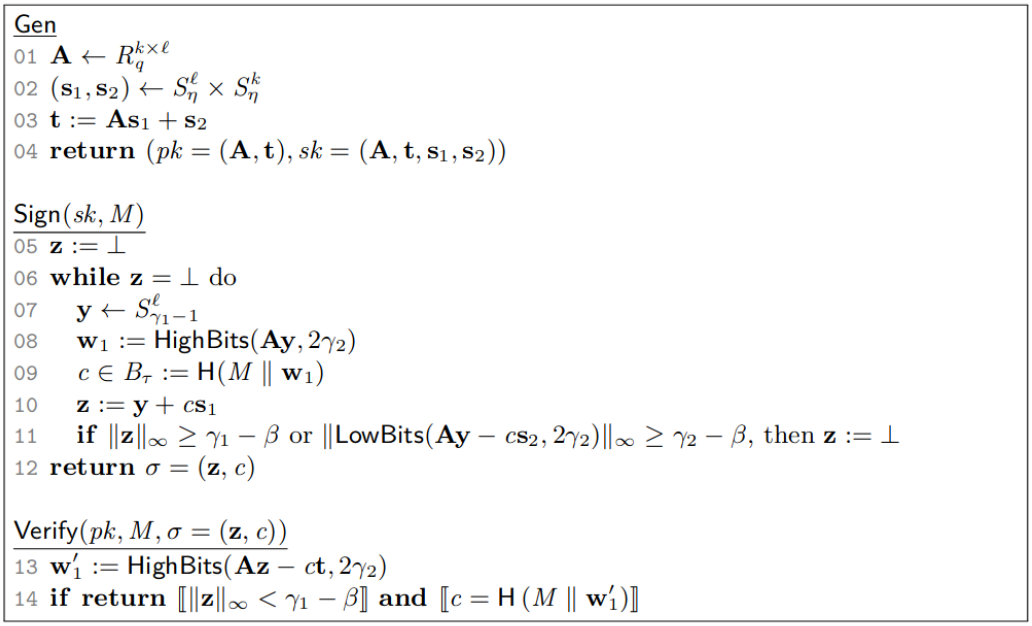

FIGURE 3. Simplified Dilitium scheme

Source: [Lyu+21, p. 4]

All operations in Dilithium are performed in the quotient ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $q = 2^{23} - 2^{13} + 1 = 8380417$ and $n = 256$. Since q is prime, $\mathbb{Z}_q = \mathbb{F}_q$ is a field. The ring is the same for all three offered security level of Dilithium. This is considered an advantage by the authors, as optimizations in one implemented level can directly be transferred to another. Most of the other parameters vary between the security levels (see [Lyu+21, Table 2, p. 15]).

For key generation, first of all a matrix $A \in R_q^{k \times l}$ is generated at random. Its entries are polynomials from $\mathbb{Z}_q[X]$ of degree less than 256. After that, two secret vectors of such reduced polynomials $(s_1, s_2) \in R_q^l \times R_q^k$ are generated. However all polynomial coefficients must be smaller than a parameter η. The public key then comprises of $pk = (A, t = As_1 + s_2)$, the private key is $sk = (A, t, s_1, s_2)$. At this point the necessity of the LWE-hardness can be seen: with a_i being the i-th row of A, and the k samples $(a_i, t_i = \langle a_i, s_1 \rangle + s_2)$, finding s_1 from given A and t means solving Ring-LWE.

The simplified signature algorithm takes sk and a message M as input. First of all, a *masking vector* $y \in R_q^l$ is randomly chosen, whose coefficients are all smaller than $\gamma_1 - 1$ for a parameter $\gamma_1 \in \mathbb{N}$ called *coefficient range*. The *low order rounding range* is defined as a second parameter $\gamma_2 \in \mathbb{N}$. It is used to decompose polynomial coefficients into "high" and "low" bits. Every $w \in \mathbb{Z}_q$ can be written as $w = w_1^* \cdot 2\gamma_2 + w_0^*$ with $|w_0^*| \leq \gamma_2$. Effectively, for w (or to be precise, the representative of the equivalence class) $w \mod \gamma_2$ is calculated, but if the result is $> \gamma_2$, the divisor is incremented and γ_2 subtracted from the remainder. This representation safes memory in comparison to the representation by the standard remainder $w_0^* = w \mod \gamma_2$ because $|w_0^*| < \gamma_2$ instead of $|w_0^*| < 2\gamma_2$. This

decomposition is done for every coefficient of every of the k polynomials in Ay, which yields the vectors w_1 with only the high bits as coefficients and w_0 with only the low bits. The message M and w_1 are concatenated and hashed with the hash function H, which has the property to produce outputs of length n containing entries 0, 1 or -1, but having exactly τ entries different from 0 for a parameter τ. The output c is called *challenge vector* and is interpreted as coefficients of a polynomial in R_q.

$z = y + cs_1$ is then a signature candidate, however the algorithm has to guarantee two conditions for a signature. Another parameter β is given, which represents the maximum possible coefficient for cs_1 and cs_2. The first condition is that all coefficients of z must be $< \gamma_1 - \beta$. Secondly, the low bits (constructed as described above) of $Ay - cs_2$ must be $< \gamma_2 - \beta$. If not both conditions are met, the procedure is repeated by the choice of a new masking vector y, otherwise the signature is (z, c). An important notice is that with this construction, Dilithium is not deterministic, i.e. the message will most likely produce different signatures when signed twice with the same private key. The actual implementation of Dilithium allows to choose between a deterministic and non-deterministic variant.

For the verification, the verifier first of all checks if z has coefficients $\geq \gamma_1 - \beta$ and returns "false" if this is the case. If not, he computes $w_1' = Az - ct = A(y + cs_1) - c(As_1 + s_2) = Ay - cs_2$ and after that the high bits of w_1' as described above. Finally, he hashes M concatenated with w_1' and compares it to the challenge vector c. He returns "true" if and only if the results are equal. This can only be the case for a valid signature: because the low bits of $Ay - cs_2$ are smaller than $\gamma_2 - \beta$ and the coefficients of cs_2 are not greater than β, the subtraction of cs_2 cannot change the high bits of Ay. Therefore, the hash function gets the same input.

3. Post Quantum Cryptography Migration of GitLab - a Case Study

Section 1 emphasized the urgent need to migrate IT systems towards PQC. Due to its rather young age as a scientific discipline, current research is so far still focusing towards

1. **Development of cryptographic primitives**, such as asymmetric encryption schemes, KEMs or Digital Signatures
    - see e.g. the submissions to the NIST PQC Standardization process ([Nat17])
2. **Reference implementations of those primitives in cryptographic software libraries**, such as
    - liboqs (see [Saf]), including code from PQClean (see [Kan+22])
    - Botan (see [Llo+23])
    - BouncyCastle (see [Leg])
3. **Performance benchmarking of reference implementations**
    - eBACS (see [BL19]) - benchmarking for many different computer architectures
    - pqm4 (see [Kan+23]) - benchmarking for microcontroller architectures
4. **Inclusion of PQC into network protocols** (see [Att+23, section 4.3] for an overview)
    - **TLS** (see e.g. [Bos+15], [SWZ16], [Why+17], [CPS19], [SKD20] as well as Open Quantum Safe's OpenSSL 3 fork (see [Saf23b]))
    - **SSH** (see e.g. [CPS19], [SKD20], [KSH23], OpenSSH (see [Pro23]: experimental support for PQC KEX in 8.0, optional since 8.9, default since 9.0) as well as Open Quantum Safe's OpenSSH fork ([Saf23a]))
    - **Media Access Control Security (MACsec)** (see e.g. [Cis20], [CS21], [Gaz+23])
    - **Internet Protocol Security (IPSec)** (see e.g. [HGL21], [Gaz+23])
    - **Internet Key Exchange Protocol Version 2 (IKEv2)** (see e.g. [Gaz+23], [Hei19])

All of the above is necessary fundamental work and delivers the building blocks of quantum-secure software. Yet, those building blocks alone are not sufficient. To reach security against the threat posed by QCs, migrations from classical to Post Quantum Cryptography within complex, multi-component, usually multi-language pieces of software are needed.

Until recently, the prevailing conviction regarding cryptographic software seemed to be that it has a rather long development cycle, consisting of regular minor patches

against reported threats, but rather rare exchanges of whole cipher suites. As stated in [MH, p. 2]), the motto was *"Set it and forget it"*. The quantum threat emphasizes the necessity to speed up this cycle and consider it a permanent duty. This principle is commonly subsumed under the term *cryptoagility*.

Cryptoagility is described as the capability to easily and swiftly locate and exchange components of cryptographic soft- and hardware (see [Hou15], [OPp19], [Wie+21] as well as [Aln+23] for a comprehensive literature review). While there is a broad consensus about the advantages for organizations being cryptographically agile, it often remains a long-term objective with an unknown path to fulfillment (see [OPp19], especially section 1.3). As illustrated by Mosca's Theorem (see [Mos15] and section 1) immediate action, however, is inevitable. PQC migrations will have to act as a field study on the way towards more cryptoagility in the future.

Currently both industry and academia have only little hands-on experience regarding the practical execution of PQC migrations exceeding the mere inclusion of PQC into network protocols. The state of affairs in the German industry has been analyzed by a common market survey of KPMG and the BSI, reflecting on the awareness of PQC ([KB23]). The response rate of only 18.3 % out of 153 contacted companies gives some indication about the importance attached to the topic. Although 95 % of the respondents assessed the risk of QCs for their cryptographic assets as "very high" or "high", only 25 % stated that this risk is properly addressed in their organization (see p. 7). With regard to Mosca's Theorem, more than 50 % of the respondents estimated y, i.e. the time necessary to complete their personal PQC migration, at more than 10 years (see Fig. 7, p. 18).

In academia, the need for further insights is reflected by the growing number of research projects related to PQC migrations. Among others, QuantumRISC ([SIT+19]), QuantumLeap ([Inf+20], German only), the NIST and US National Cybersecurity Center of Excellence (NCCoE) project "Migration to Post-Quantum Cryptography" ([SE]) and AMiQuaSy ([GgGA23], German only) ought to be mentioned. As this thesis is a contribution to the latter, the following section will give an overview of the project's problem statement and present this thesis' main goals to contribute to it.

3.1. **Problem Statement.**

AMiQuaSy (referring to the German project title *"Agile Migration auf quantenresistente Systeme"* which translates to *"Agile Migration towards quantum-resistant Systems"*, see [GgGA23]) is an industrial research project jointly conducted by Xitaso GmbH IT & Software Solutions (https://www.xitaso.com/), genua GmbH (https://www.genua.eu) and Ostbayerische Technische Hochschule Amberg-Weiden (https://www.oth-aw.de/en/). It is co-funded by the German Federal Ministry of Education and Research and covers a project period between May 2023 and April 2026.

The main goal of the project is to conduct an end-to-end PQC migration for a large, heavily used piece of end-user software, in a way that fosters cryptoagility. The lessons

learnt during this process shall be transferable to upcoming migration projects with a comparable scope.

As a migration target, the project members chose GitLab, a self-managed platform for Source Code Management ([Inc24c]). GitLab's initial purpose and main functionality is the hosting of Git software repositories, however it calls itself a "DevSecOps platform", providing many features regarding the Development, Security and Operation of software. Among others, those include Continous Integration / Continous Delivery (CD) pipelines, Code Review tools, Issue Tracker, Wikis and User Management.

According to GitLab Inc.'s information (see [Inc24d]), the platform is developed by more than 2.000 employed team members and a total of approximately 4.000 code contributors, including Open Source developers as well as developers from various other organizations using GitLab. With roughly 30 million registered users, it was ranked as the second most popular Source Code Management Platform after GitHub in the Stack Overflow Developer Survey 2022 (see [Inc22]).

Other than GitHub, GitLab's License model is Open Core: it offers a Community Edition and an Enterprise Edition with different subscription models differing in functionality, support and offered computing facilities. However, the source code enabling the main functionality is available for free under an MIT License (see LICENSE at [Inc24f]).

The source code is hosted on a public GitLab instance on GitLab.com (see [Inc24f]). It is mainly written in Ruby (using the Web Framework Ruby on Rails), as FIGURE 4 illustrates.

Language Breakdown

Language	Code Lines	Comment Lines	Comment Ratio	Blank Lines	Total Lines	Total Percentage
Ruby	2,299,620	112,636	4.7%	618,207	3,030,463	74.6%
JavaScript	730,676	18,777	2.5%	131,140	880,593	21.7%
SQL	61,018	22	0.0%	15,729	76,769	1.9%
Go	21,927	1,219	5.3%	5,203	28,349	0.7%
HTML	14,812	15	0.1%	88	14,915	0.4%
Haml	14,234	277	1.9%	4,212	18,723	0.5%
XML	6,736	4	0.1%	1,298	8,038	0.2%
shell script	3,225	524	14.0%	803	4,552	0.1%
CSS	463	38	7.6%	151	652	0.0%
Make	168	19	10.2%	34	221	0.0%
AMPL	136	0	0.0%	7	143	0.0%
Lua	14	2	12.5%	0	16	0.0%
Clojure	3	0	0.0%	0	3	0.0%
Totals	3,153,032	133,533		776,872	4,063,437	

FIGURE 4. Distribution of Programming Languages in GitLab's Source Code

Source: https://openhub.net/p/gitlab/analyses/latest/languages_summary
Retrieval Date: 17th January 2024

For reproducibility reasons, in the course of this thesis "GitLab" refers to version 16.8.0 of the GitLab Community Edition, released at 17th January 2024. From the installation instructions (see [Inc24e, section "Install/Installation methods/Self-compiled (source)"]), SOURCE CODE 1 includes the part that was executed to build it. The

repository is cloned and the necessary Ruby dependencies (called *"gems"*, see section 4) are installed.

```
1  #!/bin/bash
2  git clone -b 16-8-stable
    ↪  https://gitlab.com/gitlab-org/gitlab-foss.git
3  bundle config set --local deployment 'true'
4  bundle config set --local without 'development test kerberos'
5  bundle config path ./vendor/bundle
6  bundle install
```

SOURCE CODE 1. Install GitLab CE v16.8.0

After that, the repository encompasses an astonishing number of 39.740 Ruby files alone, as can be verified with SOURCE CODE 2.

```
1  #!/bin/bash
2  find . -type f -name '*.rb' -printf x | wc -c # Count *.rb files
```

SOURCE CODE 2. Count the number of Ruby files in the GitLab repository

Due to its large user community, its significance for many organizations as well as the Open Core license model, GitLab was a very suitable candidate for the AMiQuaSy project. Furthermore, it is to be expected that the product makes heavy use of classical cryptography to achieve various goals. The following two are given as examples:

1. The establishment of an authenticated and confidential connection between GitLab server and client uses the TLS or SSH protocol (see FIGURE 5). Those encompass a KEX such as DH or ECDH as well as Digital Signatures such as RSA, DSA or Elliptic Curve Digital Signature Algorithm (ECDSA) (see [Res18] for TLS and [LY06a] for SSH, as well as the Requests for Comments (RFCs) updating them).

2. To ensure the authenticity of commits, git is able to sign commit data with a Digital Signature (see [CS23, section 7.4]). GitLab includes the facility to verify these signatures (see [Inc24e, section "Use GitLab/Manage your Code/Repositories/Signed commits"]).

Though cryptography uses like these are identifiable from the documentation, the challenge remains not only to systematically list *all* used cryptographic assets, but especially to locate them in the source code. The following quotation from [OPp19], p.10 emphasizes this:

> *"One important challenge is identifying where public key cryptography is being used in an organization's complex IT infrastructure and which algorithms and versions are deployed. While this may seem simple, the challenge speaks to how complex and multi-layered our uses of public key cryptography are. Large organizations may leverage hundreds of software packages and develop many more. Application binaries may have chains*

of dependencies that extend to third party application providers who in turn use third party component libraries, cryptographic modules, operating system APIs, vender-specific device driver libraries, and so on. Research could help to develop active and passive approaches to scanning an infrastructure and provide an analysis of legacy cryptography usage based on network traffic, open ports, end user devices, system binaries, source code repositories, and more."

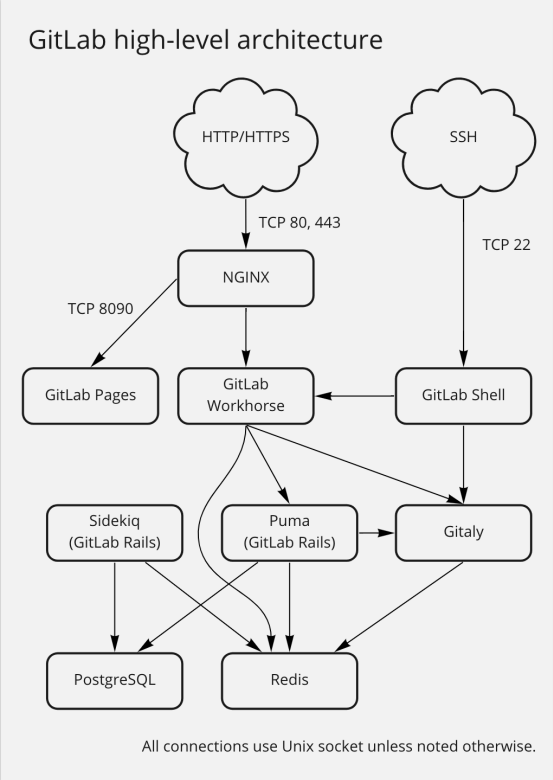

FIGURE 5. GitLab High-Level Architecture Overview

Source: https://docs.gitlab.com/ee/development/architecture.html#simplified-component-overview
Retrieval Date: 20th January 2024

The described stocktaking is a prerequisite for a thorough analysis of migration alternatives as well as the exchange of classical cryptographic primitives by PQC ones. Considering the sheer size of GitLab, its comprehensive set of features and the quantity of external dependencies it utilizes (section 4.1 handles the analysis of the used Ruby gems), the completion of an entire PQC migration under the limitations of a master thesis was not realistic. Therefore, the following limitations had to be made:

- Ruby is the language predominantly used in GitLab. The analysis is therefore restricted to Ruby Source code. Functionality written in other languages is not considered.
- Asymmetric Cryptography is at a higher risk due to QCs (see section 1). Therefore symmetric cryptography is not considered.
- It is a widely accepted Best Practice in IT security to prefer the import of systematically reviewed, actively developed and thoroughly cryptanalyzed cryptographic

assets over their custom implementation (see e.g. [Pos20], [BSW21], [Fre22], section 'Dangers and Challenges' in [Hek+23]). Due to that, the analysis is restricted to cryptography from external dependencies. Custom made cryptography is not considered.

Taking these limitations into account, the problem statement of this thesis was:

Problem Statement

Find a way to systematically locate the use of asymmetric cryptographic primitives in the Ruby gems utilized by GitLab. Evaluate the significance of the Ruby gems and choose one for migration. Compare different PQC alternatives for a replacement and choose one. Provide the PQC algorithm in the gem and ideally, integrate the gem into GitLab.

3.2. **Related Work.**

Before the solution to the problem statement could be developed, the relevant literature with regard to PQC migrations was considered. Despite a lack of practical experience, various technical and standard developing organizations published process models and handbooks with advice. Those most commonly referred to will be briefly summarized here, as their recommendations were embedded into the migration approach presented in section 3.3.

European Telecommunications Standards Institute (ETSI)

In [Ins20], ETSI proposes a migration strategy consisting of the following steps (see p. 7):

ETSI's 3 Step Approach to PQC Migrations

1. Cryptographic Inventory Compilation (CIC)
2. Migration Planning
3. Migration Execution

Hereby, the *inventory* is defined as a "set of cryptographic assets and processes in the system", while the *migration* is defined as the transition from a *non-Quantum Safe Cryptographic State* ("wherein cryptographic assets use classical, non-Quantum Safe Cryptography (QSC)") to a *Fully Quantum Safe Cryptographic State* ("of the system wherein all cryptographic assets use Quantum Safe Cryptography (QSC)" - see p. 6 for the definitions). To compile the inventory, Annex A lists guiding questions. They contain

- **Risk assessment** (Which risks arise due to the threat by QCs for your organization?)

- **Data assessment** (Which data is at risk?)
- **Cryptographic assessment** (Which cryptographic keys are in use?)
- **Infrastructure inventory** (Which cryptographic primitives, libraries, applications and certificates are in use and which administrators are in charge of them?)
- **Supplier inventory** (Which external suppliers of cryptographic assets - such as CAs or software vendors - are mandated and how reliable are they?)

As regards the inventory, also the general remarks layed down in section 4.1 and 4.2 of [Ins22] apply, even if they are not specifically formulated for the PQC migration.

For every cryptographic asset in the inventory, the migration plan shall include information about whether, when and by which quantum-safe algorithm it will be replaced and how the replacement was tested (see p. 11f.). As a general remark, [Ins20] emphasizes the advantages of cryptoagility ("Provisions for cryptographic agility should be considered [...]. Ensuring cryptographic agility will make this [the switch to a different Quantum Safe algorithm] significantly easier.", p. 12).

In 2023, ETSI applied the process model described in [Ins20] to (Cooperative) Intelligent Transport Systems consisting of autonomous vehicles. The results have been published in [Ins23].

TNO, CWI & AIVD

In [Att+23], Netherlands Organisation for Applied Scientific Research (TNO), Netherlands National Research Institute for Mathematics and Computer Science (CWI) and Netherlands National Communications Security Agency (AIVD) jointly propose their process handbook for PQC migrations. Its steps are closely aligned to the one by ETSI, namely PQC Diagnosis, Migration Planning and Migration Execution. The authors emphasize that the compilation of a cryptographic inventory should start as early as possible, as the migration itself might take "well over 10 years" (see p. 8) to be entirely completed. They underpin this estimation by claiming that the migration from SHA-1 to SHA-2 alone took "over five years" (see [Att+23, p. 6]), stressing that a later completion endangers more data from Store-now-Decrypt-later-Attacks (see p. 6f.).

The main contribution of [Att+23] is the distinction of three different *personas*, determining the recommended start and extent of actions (see section 2.1). The document gives advice on how an organization can decide which persona represents it best (see section 2.1.4) and derives particular recommendations for each persona.

Personas for PQC migrations by TNO, CWI & AIVD

1. Urgent Adopters
   - e.g. personal data handlers or providers of critical infrastructure

2. Regular Adopters
   - e.g. manufacturing companies which are not Urgent Adopters
3. Cryptography Experts
   - e.g. standard developing organisations or cryptographic infrastructure providers

As regards the migration planning, it is noted that dependencies between cryptographic assets need to be considered for a reasonable order of migration (p. 28). With respect to cryptographic agility also [Att+23] points out the associated advantages (see p. 31).

For the migration execution, the authors follow the general consensus to recommend the implementation of *hybrid solutions* (see p. 32). A cryptographic primitive is called hybrid if it combines the outcome of a classic, pre-quantum cryptographic primitive (like a Digital Signature or key from an (Elliptic Curve) Diffie-Hellman) with the outcome of a PQC primitive (see e.g. section 6.1 in [Beu+21] for hybrid methods in general, section 2.5.1 in [Pau22] and [Bin+19] for hybrid key exchanges, [Bin+17] for hybrid signature schemes). The idea is that during the transition phase, in which PQC primitives have not yet reached the maturity and cryptanalysis of classical ones, the probability of attacks and vulnerabilities is rather high. As long as QCs are not cryptographically relevant, the security of hybrid methods can then still rely on their classical component, even if the PQC one is broken.

NIST & NCCoE

NIST and the NCCoE initiated the project *"Migration to Post-Quantum Cryptography"* ([SE]) together with various collaborating organizations from both industry and academia. [BSN21] follows the general pattern of compiling a cryptographic inventory and migration plan before the execution of the actual PQC migration (see p. 5f.). Furthermore, five different scenarios with specific advices are exemplarily presented for the migration: Hardware & software modules validated according to Federal Information Processing Standard (FIPS) 140-3 (see [Sta19]), cryptographic libraries, cryptographic applications, embedded cryptography in computing platforms and cryptographic protocols. With respect to the main focus of this thesis, migrating cryptographic applications, the authors propose the following steps:

NIST & NCCoE's 6 Step Approach to PQC Migrations

1. Identification & selection of relevant applications
2. Identification of cryptographic functions inside those applications
3. Identification of used information exchange protocols
4. Identification of the operational environment

5. Compilation of algorithm characteristics (like key sizes, block sizes etc.)
6. Identification of candidates for the replacement among the submissions to the NIST PQC Standardization process (see [Nat17]).

Other Government Information Security Agencies

Several other national cybersecurity agencies have recently published their opinions about PQC migrations. The German BSI gives recommendations in [Bun20] and section 6 of [Bun21], such as increasing cryptoagility, doubling key lengths of symmetric encryption, the usage of pre-shared keys on an interim basis or hybrid solutions. However, the BSI does not propose a particular process to conduct the migration.

In [ANS22], the French National Agency for the Security of Information Systems (ANSSI) stresses the current "immaturity of PQC", because of which it will not "endorse any direct drop-in replacement of currently used algorithms in the short/medium term" (see p. 4). ANSSI is somewhat conservative in its recommendations for certain algorithms, as it tries to wait for stabilization in the standardization processes (see p. 3). Its proposed 3-phase roadmap heavily relies on the use of hybrid solutions.

ANSSI's 3 Step approach to PQC Migrations

1. Phase 1 (as of today) - Use hybrid schemes optionally, without relying on its quantum resistance (*"defense-in-depth"*)
2. Phase 2 (not earlier than 2025) - Use hybrid schemes optionally, relying on its quantum resistance
3. Phase 3 (unlikely earlier than 2030) - Use standalone PQC

3.2.1. Software Tools for Static Program Analysis.

Following the IT security principle to prefer "battle-proofed" cryptographic assets over custom implementations (see the restrictions to the problem statement layed out in section 3.1), modern software heavily relies on several well known crypto libraries such as OpenSSH ([Pro]), OpenSSL ([Com]), NaCl ([BLS]) or Bouncy Castle ([Leg]), to name a few. Moreover, also non-primarily cryptographic dependencies can make use of those crypto libraries, including them as *transitive dependencies*. Therefore, one crucial part of a cryptographic inventory is what is called "Crypto software inventory" by ETSI (see Annex A.1.4 in [Ins20]) or "Inventory of Cryptographic Dependencies" by TNO, CWI and AIVD ([Att+23], p. 23): a compilation of where which cryptographic library is used. The latter point out that ([Att+23, p. 23])

> *"Past cryptographic migrations have shown that creating an inventory of cryptographic assets is the most important and most difficult part of the diagnosis."*

The nature of this tasks begs for automation. NIST and NCCoE formulate the development of "discovery tools that can provide automation assistance in identifying where and how public-key cryptography is being used in [...] applications" ([BSN21, p. 4]) as one of their project's main goals.

A commonly proposed technique to tackle this challenge is Static Program Analysis (SPA). In a broad sense, SPA is described as scrutinizing the behavior of software without actually executing it, but by solely examining its source code (see [GS15, section 2]). Originally a tool used in Compiler Construction, it has in recent years become a versatile research area aiming to reach different goals, like e.g. type checking, coding style enforcement, program verification and security review of software (see [CW07, section 2.2]). In the context of this thesis, three different classes of SPA tools were deemed useful to support the CIC:

Classes of SPA Tools for the Cryptographic Inventory Compilation

1. **Dependency Scanner / Software Composition Analysis (SCA) Tools**
   - investigate the Software Supply Chain
   - rather fast, as they process dependencies files only (such as `Gemfiles` in Ruby or `requirements.txt` in Python)
   - output a Software Bill of Materials (SBOM)
   - search in vulnerabilities databases like Common Vulnerabilities and Exposures (CVE) (see [CH24]) for threats concerning a particular version
2. **Linter / Static Application Security Testing (SAST) Tools**
   - search for potentially dangerous / malicious patterns in code
   - rather slow, as they syntactically analyze source code files
   - output a list of possible vulnerabilities / "Code Smells"
3. **Crypto Agility Suites**
   - combination of the two classes above
   - compile a cryptographic inventory & support its management
   - often combine other IT security tools under a common interface

The compilation of a CIC asks for two different things, each of which is best solved by another tool: the construction of dependency chains for whole repositories and the syntactical analysis of source code files to find calls to cryptographic APIs. The former is a job for SCA tools, the latter for SAST tools.

Dependency chains are, among other information, included in SBOMs. A SBOM is a machine-processable file containing supply chain relationships and details of the components used in a software product (see [Sic23a, p. 7]). While their compilation is already common practice and recommended by IT security authorities like BSI (see

[Sic23a]), they rarely focus on cryptography in particular. Initiatives like IBM's proposal for a Cryptography Bill of Materials (CBOM) (see [Cor23a]) go in that direction.

Linting with SAST tools is also a well-established measure in software engineering, e.g. as part of a CI/CD pipeline. Numerous products exist that apply rule sets to scan for common undesirable or insecure code patterns. However, the sole use of classic cryptography is still too infrequently recognized as a risk factor, which is why one cannot expect default rule sets to search for it.

Crypto Agility Suites are particularly advertised by the industry as providing both functionalities. They are a rather new phenomenon following the generally increased awareness for PQC and cryptoagility. However, they are mostly proprietary software and it is hard to evaluate their quality from the provided documentation only.

Representatives of the three classes are evaluated regarding their suitability in the following section 3.3.

3.3. **Chosen Approach.**

The recommendations summarized in section 3.2 are still rather vague and generic. Furthermore, they are mainly formulated for whole organizations with complex IT infrastructures to migrate. Nevertheless, their ideas are incorporated in the approach pursued.

The presented process models reflect a common consensus which is most concisely expressed by ETSI's 3 Step Approach. NIST and NCCoE's approach can be considered a subdivision of it into six steps. Therefore, the PQC migration in this thesis is divided into cryptographic inventory compilation, migration planning and migration execution.

As shown in section 3.2.1 both ETSI, the three Dutch organizations and NIST/NCCoE agree on the usefulness of software assistance. Three different requirements are set for a tool in this respect: it must support Ruby; it must be Open Source; and it shall not only indicate the usage of a crypto library / API, but must at least be able to locate the usage in a certain source code file. Nine different tools have been analyzed regarding these requirements. The results are summarized in Table 1.

While Crypto Agility Suites like InfoSec AgileSec™ Analytics seem very promising, all available tools are proprietary software. Typical SCA tools produce SBOMs, but these alone do not pinpoint the location of usage and therefore do not satisfy the third requirement. Finally, the idea to locate calls to cryptographic APIs using SAST tools is already reflected in the literature under the term "Cryptographic API Misuse Detection" (see [Bra+19]). Eclipse CogniCrypt, CryptoLint and CryptoGo are given as inspiring examples for Java, Kotlin and Go respectively. However, the author was not able to discover a comparable solution for the Ruby environment. This led to the decision to realize the CIC by implementing and applying an own combined SCA and SAST tool, which initially constructs dependency chains targeting towards cryptographic Ruby gems and subsequently filters the Ruby files utilizing them (see section 4.1).

Class	Tool	Reference	Ruby Support	Open Source	Locate Crypto API Calls
SCA	RubySec	[Rub24]	✔	✔	✖
	BlackDuck SCA	[Inc24h]	✔	✖	✖
	Fortify Software Composition Analysis	[Lim24]	✔	✖	✖
SAST	Eclipse CogniCrypt	[Krü+17]	✖	✔	✔
	CryptoLint	[Ege+13]	✖	✔	✔
	CryptoGo	[Li+22]	✖	✔	✔
Crypto Agility Suite	InfoSec AgileSec™ Analytics	[Glo23]	✔	✖	✔
	SandboxAQ Security Suite	[San24a]	?	✖	✔
	IBM Quantum Safe	[Cor23b]	✖	✖	✔

TABLE 1. Comparison of Software Tools for Cryptographic Inventory Compilation

As regards the Migration Planning, one Ruby gem is selected as migration target. Selection criteria are the significance of the functionality used in GitLab and the Migration Execution's estimated complexity, bearing in mind the restrictions of a master thesis. The suitable PQC replacement is chosen from the finalists of the NIST PQC Standardization Process (see [Nat17]). This follows the advice given by TNO / CWI / AIVD (see [Table 4.1, p. 34][Att+23]). Among these finalists, the selection is based upon performance measures as well as the size of key pairs and the produced output.

The Migration Execution finally encompasses the provision of the chosen PQC algorithm in the Ruby gem. The fully-fledged integration into GitLab was ultimately out of scope due to time constraints, but is presented as an outlook in section 5. The implementation is thoroughly tested with unit tests.

To sum up, in the course of this thesis the following approach was pursued:

Chosen Approach for an exemplary PQC migration in GitLab

1. **Cryptographic Inventory Compilation** (see section 4.1)
   - Develop combined SCA and SAST tool to
     - construct relevant dependency chains
     - locate the usage of gems contained in the chains
   - apply the tool to the GitLab repository
2. **Migration Planning** (see section 4.2)
   - Choose a suitable gem as cryptographic asset according to the criteria
     - significance of used functionality
     - estimated complexity of Migration Execution
   - Choose a suitable PQC algorithm for replacement according to the criteria
     - must be one of the selected algorithms of the NIST PQC Standardization Process (see [Nat17])
     - performance in acknowledged benchmarks
     - size of key pairs
     - output size
3. **Migration Execution** (see section 4.3)
   - provide the PQC algorithm in the Ruby gem
   - test implementation with unit tests

This section presents the rationale and assumptions made to realize the migration approach above. Relevant architectural features of Ruby applications such as GitLab shall be briefly elaborated on.

Packaged libraries in Ruby are called *gems* (see [Shi+24, section "Libraries"]). They can be installed using the package manager RubyGems ([Qua+24]). It provides the command line interface `gem` and hosts a gem repository which lists almost 180.000 gems and counts over 150 billion downloads (see [Qua+24, "Stats"]) as of 1st February 2024.

The gems and versions needed by an application are listed in its Gemfile. Source Code 3 shows an extract from GitLab's Gemfile. It is processed by the dependency manager Bundler (see [Ark+24]), which resolves the dependencies, i.e. constructs the dependency tree and installs all transitively needed gems while avoiding version conflicts. Bundler produces a Gemfile.lock, which then lists the fixed versions of all installed gems. The example in Source Code 4 illustrates that the gem `sys-filesystem` requires `ffi` as a dependency. The Gemfile.lock is the starting point for the CIC in GitLab.

```
541  # System information
542  gem 'vmstat', '~> 2.3.0' # rubocop:todo
     ↪ Gemfile/MissingFeatureCategory
543  gem 'sys-filesystem', '~> 1.4.3' # rubocop:todo
     ↪ Gemfile/MissingFeatureCategory
544
545  # NTP client
546  gem 'net-ntp' # rubocop:todo Gemfile/MissingFeatureCategory
547
548  # SSH keys support
549  gem 'ssh_data', '~> 1.3' # rubocop:todo
     ↪ Gemfile/MissingFeatureCategory
```

Source Code 3. Extract from GitLab's Gemfile

```
1642      sys-filesystem (1.4.3)
1643        ffi (~> 1.1)
```

Source Code 4. Extract from GitLab's Gemfile.lock

4.1. **Cryptographic Inventory Compilation.**

As explicated in section 3.3, the search for a combined SCA and SAST tool trawling Ruby repositories for cryptographic API calls ended without success. Thus, the author decided to implement RubyCrypt (see [Sch24b] for the repository). Its installation and usage are documented in its `README.md`. RubyCrypt is executed running Source Code 5 in GitLab's root directory. The command line parameters are optional

and default values can be specified in `rubycrypt.rb`.

```
1  bundle exec ruby .../rubycrypt.rb SCAN_DIR EXPORT_DIR CRYPTO_GEMS
```

Source Code 5. Execution of RubyCrypt

The initial idea was the construction of a Call Graph (CG) for the whole GitLab application. A CG is a directed graph with methods as vertices and edges indicating that one method calls another during execution (see [KSK09, p. 234]). Such a graph could be filtered for vertices representing asymmetric cryptographic methods. But even if tools for the construction of Ruby applications' CGs exist, first experiments with `code-explorer` (see e.g. [Vid23]) led to the realization that this approach would require too much time and computational resources for a repository of GitLab's size. Therefore, an alternative approach was pursued.

This approach's main idea is to locate all Ruby files that utilize gems which ultimately require a cryptographic dependency. The expectation was that this approach significantly reduces the quantity of potentially relevant Ruby files from almost 40.000 to a number small enough for a manual review. The implementation was led by the following design principles:

RubyCrypt Design Principles

1. **Cryptography in Ruby is implemented in a few crypto gems.**
2. **If a gem needs cryptography, it includes a crypto gem as (transitive) dependency.**
3. **If an application needs cryptography, it requires a gem that (transitively) depends on a crypto gem.**
4. **Finding cryptographic functionality means finding the use of gems described in design principle 3.**
5. **If a Ruby file uses a gem, it contains a command that loads external code.**
6. **The loading of external code can be detected by parsing Ruby files to Abstract Syntax Trees (ASTs) and query those for loading commands.**

The first design principle has already been justified in section 3.1. The question remained: how can a list of relevant crypto gems be compiled? The compilation was based on two sources. Firstly, the Gemfile (and its authors' comments) was scanned for gems whose name or description indicate potential cryptographic functionality. Secondly, download statistics for crypto gems were considered, namely "The RubyToolbox" ([Ols24] category "Encryption & Cryptography") and "BestGems.org" ([xmi24], category "Encryption").

To make the Gemfile's scanning more convenient, RubyCrypt contains a Ruby script `gemfile_summary.rb`. After the specification of an output directory in line 17 and the script's execution in GitLab's root directory, `gemfile_summary.rb` extracts every installed gem's name and summary to a JSON file such as Source Code 6.

```
1 {
2 "rake": "Rake is a Make-like program implemented in Ruby",
3 "rexml": "An XML toolkit for Ruby",
4 "...": "..."
5 }
```

Source Code 6. Extract from a JSON file containing gem summaries

Some gems are only necessary for development purposes, testing or Kerberos support. They are marked with `group: [:development]` in the Gemfile (and / or `:test` / `:kerberos`, respectively). In the installation (see Source Code 1), these are excluded. After all, GitLab lists 328 relevant requirements in its Gemfile, and Bundler installs 506 gems in total, as printed by `bundler install`.

BestGems.org lists six gems of which two handle symmetric encryption and one is deprecated, which is why these three were excluded. From The RubyToolbox, the top five gems with regard to the number of downloads were selected. The entire selection is listed below. After the removal of duplicates, it comprises 17 distinct gems.

Chosen crypto gems

Gemfile

- attr_encrypted
- bcrypt
- cose
- doorkeeper
- encryptor
- ed25519
- json-jwt
- lockbox
- net-ssh
- net-scp
- oauth
- oauth2
- openssl
- openssl-signature-algorithm
- ssh_data

The RubyToolbox

- attr_encrypted
- bcrypt
- lockbox
- rbnacl
- scrypt

BestGems.org

- bcrypt
- cose
- rbnacl

According to the second design principle, dependency chains between gems need to be calculated. Gems can be modeled as a directed acyclic graph $\Delta = (G, D)$, where G is the set of gems and D is a set of dependency relations (g_i, g_j) signifying that gem g_i depends on gem g_j. Bundler ensures the absence of cycles during `bundle install`'s

execution. In graph theoretical terms, the search for dependency chains translates to the search for all directed paths in Δ leading to one of the 17 crypto gems. The members of a path leading to g_i are called *reverse dependencies of* g_i.

To calculate all reverse dependencies, the Bundler plugin `bundler-why` 's source code (see [Bec23]) was adapted and included in RubyCrypt . The RubyCrypt repository contains a file `reverse_dependencies.rb` . Its method `why` conducts a reverse depth-first search in Δ, as Source Code 7 illustrates.

```ruby
def traverse_specs(spec_set, parent, path = [parent])
  return if parent.nil?
  children = spec_set.select do |s|
    s.dependencies.any? do |d|
      d.type == :runtime && d.name == parent.name
    end
  end

  if children.empty?
    $relevant_paths << path.map(&:name)
  else
    children.each do |child|
      traverse_specs(spec_set, child, [child].concat(path))
    end
  end
end

# @param gem_names Strings
# @void
def why(gem_names)

  runtime = Bundler.load
  spec_set = runtime.specs # delegates to
   ↪ Bundler::Definition#specs
  $relevant_paths = []

  gem_names.each do |gem|
    spec = find_one_spec_in_set(spec_set, gem) # extracts gem
     ↪ from spec_set
    traverse_specs(spec_set, spec)
  end
  $relevant_paths
end
```

Source Code 7. Depth-first search to calculate dependency chains

The output of `why` is the array `$relevant_paths`, which contains arrays of gems. Each of these arrays represents a dependency chain, i.e. each entry is a gem depending on its successor. It is worth noting that one crypto gem can have multiple paths pointing to it and crypto gems can even depend on each other. RubyCrypt exports the dependency chains as a JSON file comparable to Source Code 8. In this extract one can see that `webauthn` transitively depends on `openssl-signature-algorithm` via two different gems, while `ssh_data` has no reverse dependencies.

```
167     [
168       "webauthn",
169       "cose",
170       "openssl-signature_algorithm"
171     ],
172     [
173       "webauthn",
174       "tpm-key_attestation",
175       "openssl-signature_algorithm"
176     ],
177     [
178       "ssh_data"
179     ]
```

Source Code 8. Extract from output file `dependency_chains.json`

The third design principle reflects the restriction to ignore custom cryptography. Following the fourth and fifth principle, RubyCrypt spots the usage of Ruby commands that load code from external sources. Ruby offers four commands to realize this (see the official Ruby Language Documentation [Shi+24], module "Kernel", section "Loading"): `autoload`, `load`, `require` and `require_relative`. If one of these is executed in a Ruby file with an argument contained in a dependency chain, this file is considered (potentially) relevant, as cryptographic functionality might have been loaded. However, also exclusively non cryptographic functionality may have been used, wich needs to be further determined by manual review.

As expressed by design principle six, searching for these four loading commands is realized by querying every Ruby file's Abstract Syntax Tree (AST). This is a common technique for SAST tools. An AST is a representation of a program as a tree, in which each node represents an operator and its child nodes represent the operators arguments, respecting the programming language's abstract grammar (see [Tur08, section 2.1]). The construction of ASTs is typically done by a parser. RubyCrypt utilizes the `parser` gem for this task (see [whi+24]), as can be seen in `parse_file.rb`. The relevant call is `ast = Parser::CurrentRuby.parse_file(path)` in line 34, where `path` is the path to a Ruby file. The traversal of the repository to find all Ruby files is implemented in `traverse.rb`.

The resulting ASTs are queried for the four loading commands. This is achieved using the `Parser::AST::Processor` class. Its method `process` traverses the AST and executes the callback `on_send` for every node that represents a method call. `on_send` is overwritten to filter for calls of a loading command and add its argument (e.g., the gem name) to an array `@required`. It ought to be mentioned that the loading commands can be used not only to load gems, but also to load other Ruby files. The relevant part of `parse_file.rb` is Source Code 9.

```ruby
class ExternalCodeFinder < Parser::AST::Processor
  @@commands = [:require, :require_relative, :load]

  def initialize
    @required = []
    super
  end

  attr_accessor :required, :path

  def on_send(node)
    # handle autoload
    if node.children[0] == nil && node.children[1] == :autoload
      @required << node.children[3].children[0]
      # handle other 3 loading commands
    elsif node.children[0] == nil &&
     ↪ @@commands.include?(node.children[1])
      @required << node.children[2].children[0]
    end
    super
  end
end
```

Source Code 9. Filter ASTs for loading commands

A pair `[path, argument]` is a *requirement*. RubyCrypt exports all requirements to a JSON file `requirements.json` structured like the extract in Source Code 10. It consists of a JSON object with one key per `argument` and an array of paths as corresponding value. Finally, the requirements are filtered for those whose `argument` contains the name of a gem which is part of at least one dependency chain (see line 52 in `rubycrypt.rb`). The filter does not match for the exact gem name only, as it is possible to include only certain classes or modules of a gem. These are specified after a `/` in the loading command. Source Code 10 illustrates the loading locations of the class `COSE::Algorithm::ECDSA` from the `cose` gem. The filtered result is exported to `filtered_requirements.json` which is structurally identical to `requirements.json`.

```
27796     "openssl/signature_algorithm/rsapss": [
27797       "./gitlab-foss/vendor/bundle/ruby/3.2.0/gems/cose-1.3.0/lib/
        ↪  cose/algorithm/rsa_pss.rb",
27798       "./gitlab-foss/vendor/bundle/ruby/3.2.0/gems/
        ↪  openssl-signature_algorithm-1.3.0/lib/openssl/
        ↪  signature_algorithm.rb"
27799     ],
27800     "cose/algorithm/ecdsa": [
27801       "./gitlab-foss/vendor/bundle/ruby/3.2.0/gems/cose-1.3.0/lib/
        ↪  cose/algorithm.rb"
27802     ],
```

SOURCE CODE 10. Extract from output file `requirements.json`

Altogether, for every execution RUBYCRYPT produces a timestamped output directory structured as in FIGURE 6. `config.json` contains the path to the scanned directory as well as the crypto gems searched for. `errors.json` and `error_log` contain human-readable error messages. `config`, `dependency_chains`, `errors`, `requirements` and `filtered_requirements` are binary files whose content is identical to the corresponding `.json` files - they can be loaded as hashes in Ruby using the `analysis.rb` script for further analysis.

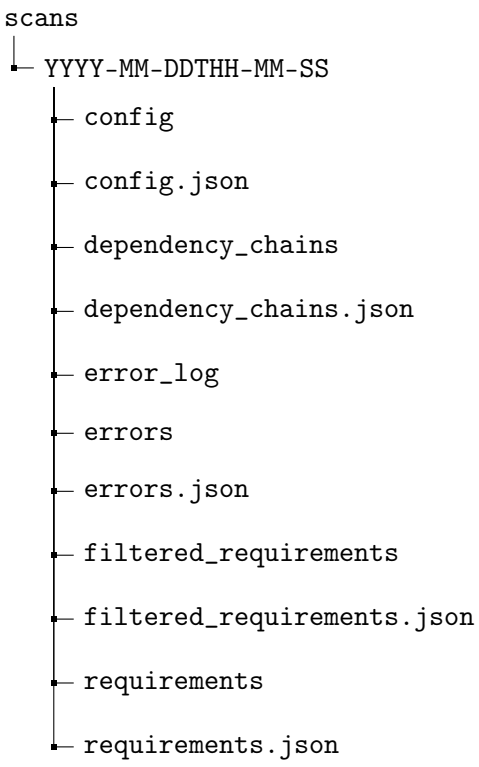

FIGURE 6. Structure of output directory produced by RUBYCRYPT

4.1.1. **Results.**

This section describes the results produced by running RubyCrypt on GitLab's source code. The output files are accesible in the `results_gitlab_scan` branch of the RubyCrypt repository (see [Sch24b]), which also contains a `gem_summaries.json` and a `not_from_gem_source.json`, the meaning of which is explained below.

To analyze the results, the `analysis.rb` script is provided. It prompts for the scan subdirectory and outputs some statistics, as Source Code 11 illustrates. When loaded in an interactive Ruby interpreter such as `irb`, it is possible to further investigate the data stored in global variables corresponding to the file names, for example `$filtered_requirements`.

```
1  #!/bin/bash
2  ruby analysis.rb
3  Enter timestamp of output directory
4  2024-01-25T18-47-07
5
6  DEPENDENCY CHAINS
7
8  No. of searched crypto gems:                  17
9  |----found in scan directory:                 15
10 No. of transitive dependencies:               44
11
12 REQUIREMENTS
13
14 No. of distinct loading arguments:            10222
15 |----which are gems from dependency chains:   598
16     |----which are not from gem source code:  8
17
```

Source Code 11. Statistics about RubyCrypt results produced by `analysis.rb`

Out of 17 crypto gems, 15 were indeed found in GitLab - all but `rbnacl` and `scrypt`. The unique union of all dependency chains contains 44 gems, which transitively depend on these 15 crypto gems.

`requirements.json` contains 10.222 distinct loading command arguments, of which 598 are gems from the dependency chains. All 44 gems from the dependency chains are loaded. As mentioned above, the arguments do not only contain gems, but also paths to other Ruby files. According to design principle three from section 4.1 these can be ignored, as only gems are sought. One pattern was immediately noticeable in the filtered requirements: the vast majority of them contains paths in the `vendor/bundle` subdirectory. To be precise, only eight loading arguments are not located there.

`vendor/bundle` is the directory into which Bundler puts the source code of the installed gems (see [Ark+24, "Doc/bundle install/Deployment Mode" Nr. 3]). By far

the most requirements are therefore not written in the GitLab application itself, but its dependencies. This was indeed unexpected by the author. It raised the question as to how GitLab actually imports its dependencies. As it turned out, design principles five and six represent an assumption which must be rejected: the loading of a gem does *not* necessarily have to be specified in the very location of its usage. This has three main technical reasons:

1. GitLab is not only a Ruby application, but is based upon the web framework Rails ([Fou24b]). Rails Applications use `Bundler.require` to globally require all gems from the Gemfile.lock ([Wei], [Shi+24, module `Bundler`, method `require`]). This can be seen in line 18 in `gitlab-foss/config/application.rb`. Due to that, the loading and usage of gems are in separate locations.

2. `autoload` provides *Lazy Loading* in native Ruby. Receiving a constant and a loading target, it registers the loading target to be loaded the first time the constant is accessed (see [Shi+24, module `Kernel`, method `autoload`]). The constant can be a class of a gem, which shall be loaded lazily. The `autoload` command does not necessarily have to be located in the same file as the constant's first use, but only needs to be executed before it.

3. Rails offers even more capabilities for auto- / lazy loading (see [Fou]).

These features, though convenient for the developer, make SPA as intended by RubyCrypt nearly impossible. Apart from that, two more challenges ocurred. Firstly, some arguments do not just contain gem names but additional logic to specify the loading target. RubyCrypt does not execute the program logic to evaluate the actual loading argument. An example from `gitlab-foss/gems/gitlab-housekeeper/lib/gitlab/housekeeper/runner.rb` in Source Code 12 and the corresponding key in Source Code 13 demonstrate this:

```
58              require(Pathname(r).expand_path.to_s)
```

Source Code 12. Requirement with program logic

```
13363   "(send\n  (send nil :Pathname\n    (lvar :r)) :expand_path)": [
13364     "./gitlab-foss/gems/gitlab-housekeeper/lib/gitlab/
        ↪ housekeeper/runner.rb"
13365   ],
```

Source Code 13. Key of a requirement with program logic in `requirements.json`

The second challenge regards the encoding of Ruby files. The `parser` gem only accepts Ruby files encoded in 8 Bit Unicode Transformation Format (UTF-8) and raises a `Parser::SyntaxError` for characters with other encodings, such as 16 Bit Unicode Transformation Format (UTF-16). `errors(.json)` contains the paths of the files with occurred errors, further error details are redirected to `error_log`. Almost all `Parser::SyntaxError` regard test files safeguarding these errors' proper handling.

To sum up, RubyCrypt's approach turned out to be too simplistic for applications using the advanced concepts mentioned above, such as GitLab. Yet section 5 identifies use cases for RubyCrypt and presents potential improvements and mitigation for its shortcomings. Even if a complete cryptographic inventory could not be constructed, the results of RubyCrypt were utilized to carry on with the migration planning.

4.2. **Migration Planning.**

Source Code 14 shows the eight loading arguments which are part of the dependency chains, but do not originate from gem source code in `vendor/bundle`.

```
{
  "devise/orm/active_record": [
    "./gitlab-foss/config/initializers/8_devise.rb"
  ],
  "openssl": [
    "./gitlab-foss/config/initializers/excon.rb",
    "./gitlab-foss/lib/gitlab/ci/ansi2json/state.rb",
    "./gitlab-foss/lib/gitlab/email/smime/signer.rb",
    "./gitlab-foss/lib/gitlab/x509/commit.rb",
    "./gitlab-foss/lib/gitlab/x509/signature.rb",
    "./gitlab-foss/lib/gitlab/x509/tag.rb",
    "./gitlab-foss/lib/omni_auth/strategies/jwt.rb"
  ],
  "rack/oauth2": [
    "./gitlab-foss/lib/api/api_guard.rb"
  ],
  "ed25519": [
    "./gitlab-foss/lib/gitlab/ssh/signature.rb"
  ],
  "omniauth-oauth2": [
    "./gitlab-foss/lib/omni_auth/strategies/bitbucket.rb"
  ],
  "../../config/initializers/doorkeeper": [
    "./gitlab-foss/spec/initializers/doorkeeper_spec.rb"
  ],
  "webauthn/fake_client": [
    "./gitlab-foss/spec/services/webauthn/
↪ authenticate_service_spec.rb",
    "./gitlab-foss/spec/services/webauthn/register_service_spec.rb",
    "./gitlab-foss/spec/support/helpers/fake_webauthn_device.rb"
  ],
  "devise_helpers": [
    "./gitlab-foss/spec/support/helpers/login_helpers.rb"
  ]
}
```

Source Code 14. Filtered Requirements which are not from gems' source code

The keys `../../config/initializers/doorkeper`, `webauthn/fake_client` and `devise_helpers` are all loaded in the `gitlab-foss/spec` subdirectory. Ruby applications' `spec` directories contain test cases (or *specifications*) which are evaluated with testing frameworks such as RSpec or minitest. Findings in this directory are considered irrelevant.

The descriptions from `gem_summaries.json` as well as further manual code review revealed that the remaining five gems or classes provide two cryptographic functionalities: user authorization based on the OAuth 2.0 protocol (see [Par]) and the verification of git commit signatures, which was already mentioned in section 3.1. `devise/orm/active_record`, `rack/oauth2` and `omniauth-oauth2` are used to enable the first, while the requirements of `openssl` and `ed25519` point toward Ruby files realizing the second. As determined in the chosen approach in section 3.3, the selection of a migration target is based upon the criteria significance of functionality as well as expected migration complexity.

Significance can be evaluated with respect to the implications and severity of a possible attack. In this regard, user authorization is considered more significant. The impersonation of another GitLab user harbours dangers such as misconfiguration of the account, deletion of repositories or the retrieval of user secrets. Breaking the signature algorithm used for commit signing, e.g. by gaining access to a private RSA key through factoring with Shor's algorithm, would indeed enable an attacker to impersonate another committer. It would therefore ruin the authenticity of code contributions. Yet, the severity of a broken user authorization outweighs this.

As regards complexity, the expected migration of commit signature verification is deemed considerably less intricate in comparison to user authorization. GitLab offers no less than 20 different OAuth providers, i.e. accounts form other platforms which can be used as credentials to login to GitLab (see [Inc24e, section "Administer/Authentication and authorization"]). This is also reflected in the `dependency_chains.json`: 17 out of 37 chains point to `oauth` or `oauth2`, in total they combine 20 out of all 44 reverse dependencies. During a migration, the interplay between these 20 and `oauth` / `oauth2` needs to be taken into account. Furthermore, OAuth 2.0 is itself a sophisticated protocol standardized by several RFCs, which must be considered (see [Par, section "Specs"]). Commit signatures, on the other hand, are a clearly delimited, stand-alone functionality which utilize a single cryptographic primitive, i.e. digital signatures. It was therefore reasonable to initiate the exemplary PQC migration with the commit signature verification.

Commit signing is done by git, not GitLab. It allows to safeguard that the originator of a commit is indeed someone listed in an `allowed_signers` file. Which string of commit data is signed can be seen in the method `commit_tree_extended` of the git repository's `commit.c` file (see [git24]). Source Code 15 displays the structure of such a string. The commit data, together with its signature, can be printed by executing `git cat-file commit COMMIT_HASH`. git allows three different types of key formats for commit signing: SSH keys ([LY06a]), GNU Privacy Guard (GPG) keys ([gCod24]) and keys from X.509 certificates ([Uni19]). All three support different primitives such

as RSA or ECDSA. Signing and verifying are implemented in git's `gpg-interface.c`. It provides methods `sign_buffer_ssh` and `verify_ssh_signed_buffer` for SSH keys as well as `sign_buffer_gpg` and `verify_gpg_signed_buffer` for GPG and X.509 keys.

```
1  tree TREE_HASH
2  parent PARENT_COMMIT_HASH
3  author NAME EMAIL COMMIT_UNIX_TIMESTAMP TIMEZONE_OFFSET
4  committer NAME EMAIL COMMIT_UNIX_TIMESTAMP TIMEZONE_OFFSET
5
6  COMMIT_MESSAGE
```

Source Code 15. Commit data signed by git

To be able to display and process the verification status of a signed commit, GitLab itself also implements the verification functionality. According to the three key formats, GitLab defines modules `Ssh`, `Gpg` and `X509` in the `lib/gitlab/ssh|gpg|x509` sub-directories, to which RubyCrypt pointed. `Commit` and `Signature` classes are defined in the `Ssh` and `X509` modules in the associated `commit.rb` and `signature.rb` files, `Gpg` only defines a `Commit` class.

The `X509::Signature` class utilizes the `openssl` module for signature verification. It wraps the C library OpenSSL ([Com]). As `openssl` is part of the Ruby core library, the provision of a quantum-secure digital signature algorithm would be quite cumbersome. Other than simply updating a gem, the code providing the digital signature algorithm would need to be accepted for a merge in the Ruby core library and its provisioning would require the release of a new Ruby version. This process could take a long time and would depend on the prioritisation of new language features by the Ruby language maintainers. Therefore, `openssl` was not considered as a migration target.

The `Gpg::Commit` class uses the `gpgme` gem, `Ssh::Signature` the `ssh_data` gem for signature verification. Neither gem has reverse dependencies and both are considered equally relevant. With respect to the expected migration execution's complexity, it was guessed that the implementation into `ssh_data` could profit from the availability of the OpenQuantumSafe project's OQS-OpenSSH fork ([Saf23a]), e.g. for the generation of test instances. Therefore, the choice for the migration target fell on `ssh_data` ([Inc24b]). Though not precisely relevant for GitLab as such, the following fact shall be mentioned, which is not without a certain irony: out of all organizations, it is GitLabs biggest competitor GitHub providing `ssh_data` and also using it for commit signature verification in its products (see [Inc24a, section "Authentication/Verify commit signatures/Commit signature verification/SSH commit signature verification"]).

Without git's ability to actually use Dilithium for commit signatures, the whole migration of `ssh_data` would be somehow pointless. Due to that, as part of migrating the whole commit signature functionality, Dilithium shall also be included into git itself.

The migration planning's last step is the selection of a PQC digital signature algorithm. As set out in the chosen approach, the choice was between the three finalists of the

NIST PQC Standardization Process: CRYSTALS-Dilithium ([Lyu+]), Falcon ([Fou+]) and SPHINCS+ ([Hül+]). NIST defined five different security strength categories for its challenge (see [Nat17, "Call for Proposals", section 4.A.5]). All three finalists are available with different parameter sets for different security strength categories. For comparability reasons, only parameter sets of the highest category 5 were considered, namely Dilithium5, Falcon-1024 and SPHINCS+-SHA2-256s-simple. TABLE 2 displays relevant size measures in bytes.

Criteria	**CRYSTALS-Dilithium**	**Falcon**	**SPHINCS+**
Branch of PQC	lattice-based	lattice-based	hash-based
Public Key Size	2592	1793	64
Private Key Size	4864	2305	128
Signature Size	4595	1280	29792

TABLE 2. Key and signature sizes of PQC Digital Signature Algorithms in bytes

As regards performance, results from the benchmarking project eBATS (ECRYPT Benchmarking of Asymmetric Systems, see [BL19]) are considered. In eBATS, different signature algorithms are benchmarked in different implementations on different computer architectures. FIGURE 7, 8 and 9 display the median number of Central Processing Unit (CPU) cycles necessary to sign a message after several repetitions. Two different implementations are compared: one that uses the CPU instruction set extension Advanced Vector Extensions (AVX) 2 and one that does not.

SPHINCS+ is generally considered a very conservative choice, as the idea of constructing signature schemes from secure cryptographic hash functions goes back to the beginning of public key cryptography and is therefore very well understood and cryptanalyzed (see [Beu+21, p. 23 f.]). Yet although its key sizes are significantly smaller, SPHINCS+ signature size exceeds those of Falcon and Dilithium by far. Furthermore, it is the slowest of the three candidates. These reasons are considered criteria for the exclusion of SPHINCS+.

Between Falcon and Dilithium, there are only negligible differences regarding the metrics under consideration. Falcon has slightly smaller key and signature sizes, while Dilithium is somewhat faster. The decision was therefore taken in consideration of another factor: as stated in [Beu+21, p. 20], the high complexity of the underlying construction in Falcon might lead to implementation vulnerabilities in its reference implementations. So ultimately, Dilithium was chosen.

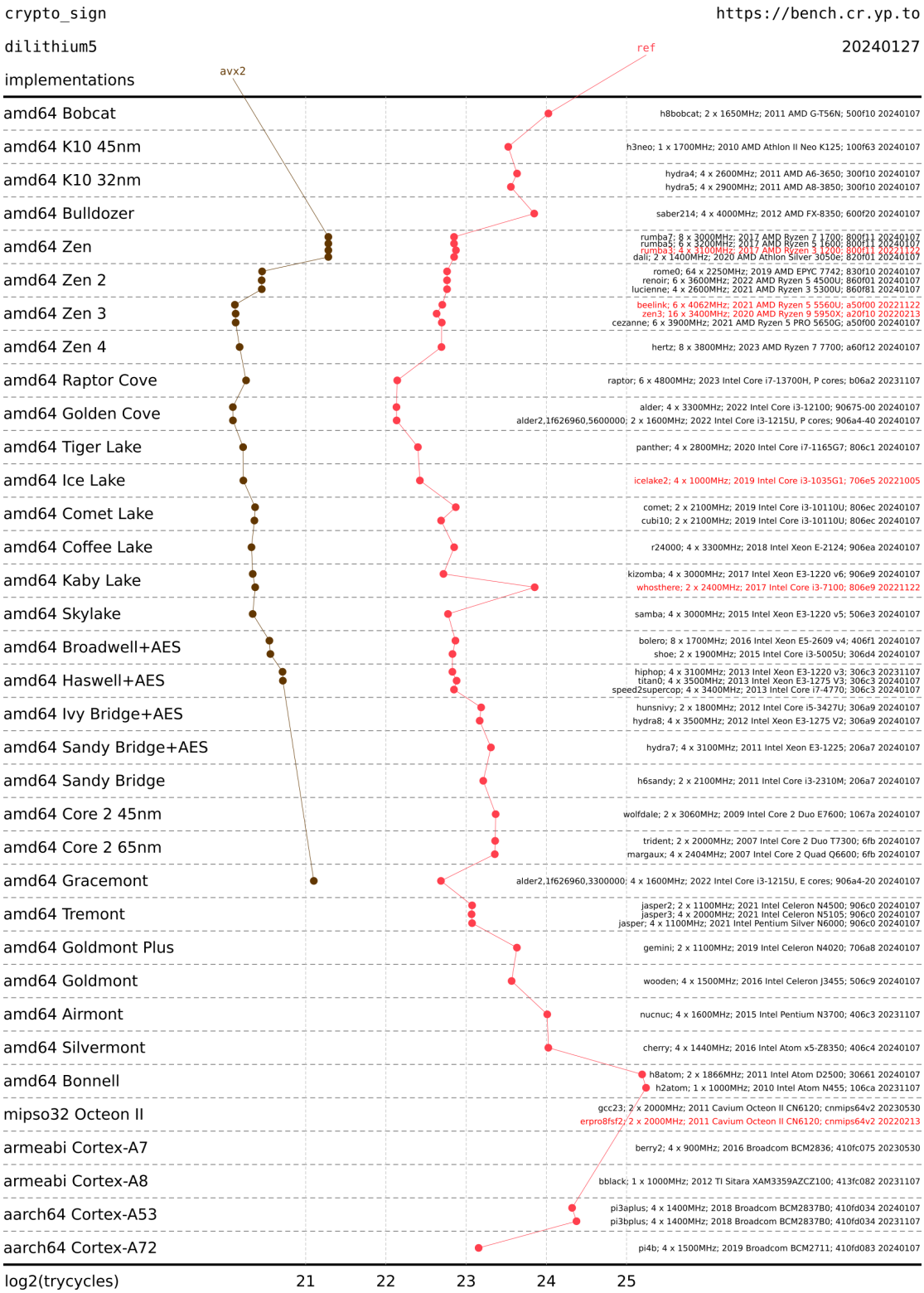

FIGURE 7. Dilithium5 performance benchmarking for different architectures

Source: https://bench.cr.yp.to/impl-sign/dilithium5.html
Retrieval Date: 4th February 2024

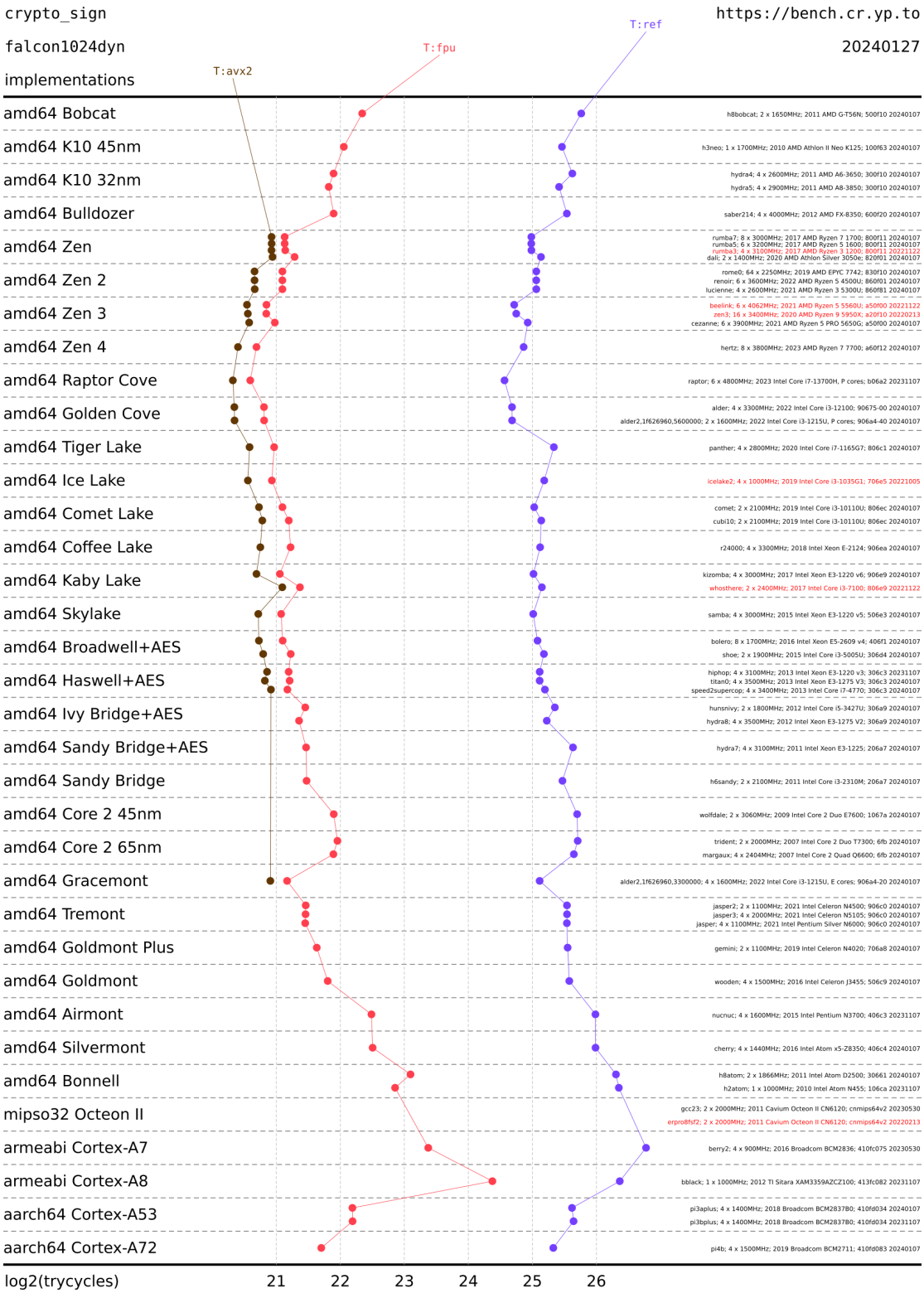

FIGURE 8. Falcon-1024 performance benchmarking for different architectures

Source: https://bench.cr.yp.to/impl-sign/falcon1024dyn.html
Retrieval Date: 4th February 2024

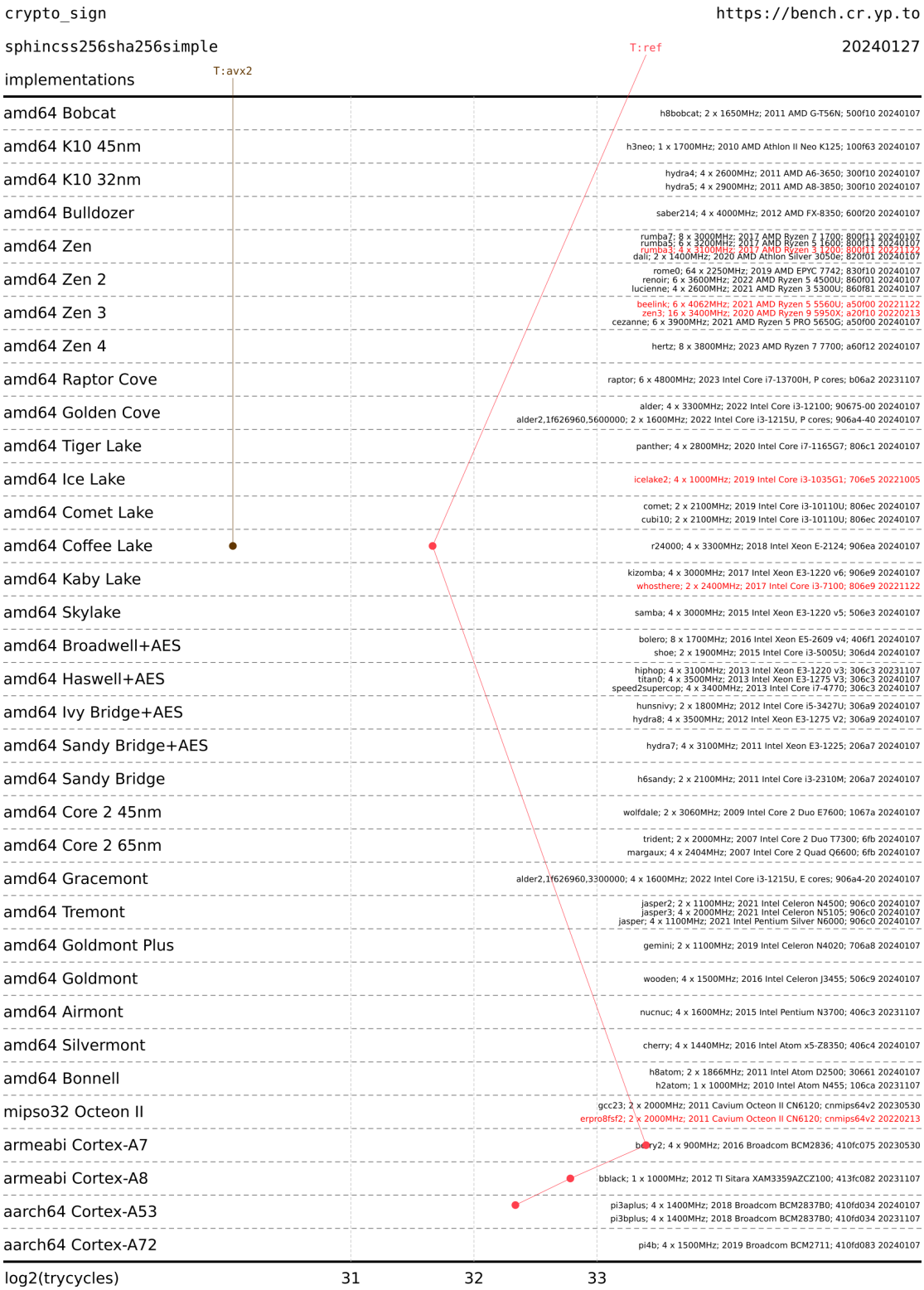

FIGURE 9. SPHINCS+-SHA2-256s-simple performance benchmarking for different architectures

Source: https://bench.cr.yp.to/impl-sign/sphincss256sha256simple.html
Retrieval Date: 4th February 2024

4.3. **Migration Execution.**

The migration planning gave rise to two objectives: (i) enabling git to produce Dilithium commit signatures and (ii) providing Dilithium support for SSH key pairs, certificates and signatures in `ssh_data`. After a brief presentation of necessary preliminaries regarding key generation and data signing in OpenSSH ([Pro]), section 4.3.1 presents a realization of the first objective. To the best of the author's knowledge, this has not yet been demonstrated elsewhere. After that, section 4.3.2 describes the inclusion of Dilithium5 to `ssh_data` .

4.3.1. **PQC Commit Signatures in git.**

Originally, the SSH protocol exclusively used key pairs to provide a method of user authentication between client and server ("Host-Based Authentication", see [LY06b, section 9.4.6], [LY06a] and updating RFCs). As of version 8.1, the protocol's most common implementation OpenSSH adds the functionality of using SSH key pairs to sign arbitrary data (see [Pro23]). To generate these key pairs, OpenSSH offers the command `ssh-keygen -t KEYTYPE`. As can be seen in `ssh-keygen`'s man page, the available key types in version 8.9p1 are `dsa | ecdsa | ecdsa-sk | ed25519 | ed25519-sk | rsa`, i.e. only classical ones. Apart from offering one hybrid method as KEX for the session establishment (`sntrup761X25519-sha512@openssh.com`), OpenSSH does not yet provide non-experimental support for other PQC methods up to the latest version available to the time of writing (v9.6).

To include PQC methods to OpenSSH, the Open Quantum Safe project team developed its OQS-OpenSSH fork ([Saf23a]). With this fork, one can generate keys for different variants of Dilithium, Falcon and SPHINCS+, including Dilithium5 (for a complete list see [Saf23a, `README.md` section "Digital Signature"]). These key types can then be used for signing and verifying git commit data, if git is configured as follows. For the example, a Dilithium5 key pair is used.

First of all, the fork has to be installed before a Dilithium5 key pair can be generated. An identification string for the committer (typically, but not necessarily the e-mail address configured in git) and the public key need to be added to an `allowed_signers` file, which lists the trusted committers. The script in Source Code 16 realizes these two steps. By altering the argument of the `-t` flag in line 3, another key type can be chosen.

```
1  #!/bin/bash
2  # generate Dilithium5 key pair; write to id_dilithium5(.pub)
3  /PATH/TO/OQS-OpenSSH/ssh-keygen -t ssh-dilithium5 -f id_dilithium5
4  # append e-mail and public key to allowed_signers
5  echo $(git config user.email;cat id_dilithium5.pub) >>
   ↪ allowed_signers
```

Source Code 16. Generate SSH key pair & append to git's `allowed_signers`

After this preparation, git must be configured properly. git offers three scopes for setting configurations: system, user and repository. For every scope there is a configuration file which contains the settings (see [CS23, section 8.1]). Source Code 17 shows a working example of a configuration file allowing to use a Dilithium5 key for commit signing. [Sch24a] includes a repository with this configuration.

```
1  [core]
2    repositoryformatversion = 0
3    filemode = true
4    bare = false
5    logallrefupdates = true
6  [gpg]
7    format = ssh
8  [user]
9    signingKey = /path/to/id_dilithium5
10 [gpg "ssh"]
11   allowedSignersFile = /path/to/allowed_signers
12   program = /path/to/OQS-OpenSSH/ssh-keygen
13 [commit]
14   gpgSign = true
```

Source Code 17. `.git/config` file enabling Dilithium commit signatures

Line 14 is optional: it enforces a signature for every commit, yet it is also possible to use the `-S` flag together with `git commit` to sign a single commit. The verification status of commit signatures can then e.g. be verified with `git log --show-signature`. Source Code 18 shows an exemplary output. One can see that git utilized Dilithium5 for the commit signature.

```
1  commit 140e0979d2ced24f0a2aa42f0e7c44970e4b9127 (HEAD -> main,
   ↪  origin/main)
2  Good "git" signature for florian.schroeck@stud.htwk-leipzig.de
   ↪  with DILITHIUM5 key
   ↪  SHA256:/eWeEGZ5dKVMDu7AJg/yX2NnhLqVid1i+5ygxcY2lgc
3  Author: flck <florian.schroeck@stud.htwk-leipzig.de>
4  Date:   Fri Jan 19 18:32:01 2024 +0100
5
6          Change README
```

Source Code 18. Verification of Dilithium commit signature in git

4.3.2. Including Dilithium to `ssh_data`.

`ssh_data`'s summary reads *"Library for parsing SSH certificates"*. In fact, the gem provides more features, i.e. the parsing and SSH-specific encoding of public keys, signatures and certificates, the parsing of private keys as well as data and certificate signing using private keys and signature verification using public keys. As regards the algorithm

types, it provides support for DSA, ECDSA, ed25519, RSA and the hardware security key variants ECDSA-SK and ed25519-SK, just as OpenSSH does. The objective of `ssh_data`'s PQC migration is to offer the same functionality for Dilithium5 keys.

First of all, a Dilithium implementation available in Ruby needed to be found. All reference implementations submitted to the NIST PQC Standardization are written in C or Assembly, so are not natively embeddable to `ssh_data`. Bearing in mind cryptoagility, it is reasonable to include a generic crypto library instead of the stand-alone implementation of one algorithm, to ease its exchange at a later point in time.

Three major crypto libraries with Ruby language bindings were considered: Botan ([Llo+23]), Bouncy Castle ([Leg]) and liboqs ([Saf]). All three offer gems for Ruby support, namely `ruby-botan` ([Inc20]), `bouncycastle` ([Nel19]) and `roqs` ([Lia23]). However, `ruby-botan`'s last commit was made in 2019 and other than the C library, it does not yet include Dilithium, Falcon or SPHINCS+. The same holds true for `bouncycastle`, which wraps release 1.70 of its corresponding Java library, while the three PQC signature algorithms are not supported before release 1.72. Therefore, `roqs` was selected, which wraps the C library liboqs (see section 3). Yet another reason in favour of `roqs` is the fact that OQS-OpenSSH also uses liboqs, so the syntax in `ssh_data` would be comparable.

One drawback of liboqs is that it only provides primitive PQC signatures, but does not yet offer hybrid ones. Those are already implemented in OQS-OpenSSH: for algorithms in NIST security strength category 5 (like Dilithium5), a combination with an ECDSA signature using the nistp521 elliptic curve is available (see [Saf23a, `README.md` section "Digital Signature"]). The provision of this in `ssh_data` would have required the development of a Ruby wrapper around the OQS-OpenSSH C library, while the wrapper `roqs` around liboqs was readily available.

While, as set out in section 3.2, it is generally recommended to implement hybrid solutions, there is not yet a consensus about which of the various *combiners* of classical and PQC signatures is to be preferred. [Bin+17] present different combiners and analyze them for the security notion of *unforgeability under chosen message attacks* (EUF-CMA), i.e. the ability of an adversary to obtain a forgery on a signed message under the assumption of having access to an oracle producing signatures on randomly chosen messages. However, even very recent publications such as [Ber+22b, section 4.3] or [BH23] still do not provide a clear recommendation on a particular combiner. Hence it is not even secure if, once git includes PQC signature algorithms, its maintainers will choose the same combiner as OQS-OpenSSH. In any case, having at hand the primitive PQC signature, all combiners can be constructed. For these reasons, the migration in the course of this thesis did not include the implementation of a hybrid signature in `ssh_data`, but rather leaves it as future work (see section 5).

To conduct `ssh_data`'s migration, initially the repository provided by GitHub was forked (see [Sch24c, branch "include-post-quantum-crypto"]). All changes made by the author can be listed by executing Source Code 19, which compares the current state of the repository with its state at the time of forking.

```
1  git diff HEAD 4ec1875211a744da22071afeb9c37eb216e011fd
```

Source Code 19. Show all changes to `ssh_data`

The migration consisted of the following steps:

`ssh_data` Migration steps

1. Add `roqs` dependency
2. Enable generation and parsing of Dilithium private and public keys as well as encoding of public keys
3. Enable parsing, generation and encoding of SSH certificates
   - including Dilithium public keys
   - signed with Dilithium private keys
4. Enable parsing and generation of Dilithium SSH signatures
5. Create Dilithium fixtures for unit tests
6. Write unit tests for all added functionality
7. Complement documentation in `README.md`

For step one, `roqs` was added as a dependency to `ssh_data`'s gem specification, as can be seen in line 15 of `ssh_data.gemspec`. As the Gemfile points to `ssh_data.gemspec`, `roqs` is installed when `bundler install` is executed.

The SSH key pairs' data formats are defined in [LY06c, section 6.6] and the updating RFCs, referencing data types defined in [LY06b, section 5]. Even if the RFCs specify the encoding of both public *and* private keys, `ssh_data` does not implement the encoding of private keys. Therefore, it was also omitted for Dilithium. Firstly, the subclasses `PublicKey::Base::DILITHIUM` and `PrivateKey::Base::DILITHIUM` were created implementing the necessary methods. As regards the private key subclass, these are the constructor `initialize`, the key generation method `generate` calling the attribute setter `from_liboqs` as well as `sign` and `cleanup`. For key generation, `generate` uses the `Roqs::SIG.genkeypair` method from `roqs`, which ultimately calls the underlying method form the C library liboqs. This is why it returns the keys as a pair of `Fiddle::Pointer`, the Ruby class to handle C pointer. This made it necessary to implement `cleanup`: the method overwrites the key pointer's memory location with zeros before freeing the pointer to prevent the involuntary disclosure of private keys through reading memory by an adversary. Unsurprisingly, `sign` calls the underlying `roqs` method `Roqs::Sig.sign`.

The public key subclass contains the constructor `initialize` as well as `verify`, an instance comparison method `==` and `rfc4253`. The latter provides the encoding. To facilitate the parsing and encoding, the `Encoding` module needed to be expanded. This comprised the definition of private and public key components in `DILITHIUM_PRIVATE_KEY_FIELDS` and `DILITHIUM_KEY_FIELDS` as well as the implementation of `encode_int8_array_pointer` and `decode_int8_array_pointer`. As the name suggests, they carry out the de- and encoding of the key values stored at the memory addresses in the corresponding `Fiddle::Pointer`. To complete migration step two, the superclasses' universal constructor `from_data` were extended to parse Dilithium SSH key data. Providing that a Dilithium5 key pair was generated with OQS-OpenSSH, Source Code 20 demonstrates its handling in `ssh_data`.

```ruby
require 'ssh_data'
# --- Parsing ---
# Read private key file
key_data = File.read("id_dilithium5")
# Create PrivateKey / PublicKey::Base::DILITHIUM object
private_parsed = SSHData::PrivateKey.parse_openssh(key_data).first
public_parsed = private_parsed.public_key

# --- Generation ---
private_gen = SSHData::PrivateKey::DILITHIUM.generate
public_gen = private_gen.public_key

# --- Encoding ---
public_gen.openssh

# --- Signing ---
raw_sig = private_gen.sign("message")

# --- Verification ---
public_gen.verify("message", raw_sig) # true
public_gen.verify("another message", raw_sig) # false
```

Source Code 20. Handling of Dilithium5 keys with `ssh_data`

With the preparation in step one and two, step three was limited to specifying the SSH algorithm identifier for Dilithium from OQS-OpenSSH (`ssh-dilithium5`) in `certificate.rb`. Source Code 21 illustrates Dilithium5 certificate handling in `ssh_data`.

```ruby
require 'ssh_data'
# --- Parsing ---
# Read certificate from file
cert_data = File.read("path/to/dilithium.cert")
```

```
# Create Certificate object
cert_parsed = SSHData::Certificate.parse_openssh(cert_data).first

# --- Generation ---
cert_gen = private_gen.issue_certificate(
  public_key: public_gen,
  key_id: "my-ident"
  # further certificate data can be specified
)

# --- Encoding ---
cert_gen.openssh

# --- Verification ---
cert_gen.verify # true
```

SOURCE CODE 21. Handling of Dilithium5 certificates with `ssh_data`

Indeed, step four did not require any further code adaption, as the signing was already implemented in step two and the parsing is identical for all algorithm types. Given that OQS-OpenSSH was used to write a Dilithium5 signature of the string `message` into the signature file `data.sig`, SOURCE CODE 22 demonstrates signature handling in `ssh_data`.

```
require 'ssh_data'
# --- Parsing ---
# Read signature file
sig_data = File.read("data.sig")
sig_parsed = SSHData::Signature.parse_pem(sig_data)

# --- Generation ---
sig_gen = private_gen.sign("message")

# --- Verification ---
sig_parsed.verify("message") # true
sig_parsed.verify("another message") # false
```

SOURCE CODE 22. Handling of Dilithium5 signatures with `ssh_data`

To produce test fixtures, `ssh_data` contains two Bash scripts utilizing OpenSSH: `gen.sh` for key pairs and certificates, `create-signatures.sh` for signatures. Both have to be executed in the directory of their location. To generate Dilithium objects, standard OpenSSH has to be replaced by the OQS fork. Consequently, both scripts were adapted to prompt for the fork's path. `gen.sh` now generates SSH certificates containing classical keys and signed by a Dilithium key. Certificates including Dilithium keys cannot yet be handled by OQS-OpenSSH, but in fact `ssh_data` can now handle them. The

generation of these certificates is implemented in the test file `certificate_spec.rb` using `ssh_data` itself. This way, even if `ssh_data`'s correctness cannot be tested against OQS-OpenSSH, at least the cycle of certificate generation, parsing and verification can be tested. Finally, `create-signatures.sh` was extended to sign a randomly generated message with a Dilithium private key, both within a certificate and not.

Step six was based on the already existing test cases for the classical algorithm types: each existing test case was adapted to Dilithium in the `..._spec.rb` test files. Again, OQS-OpenSSH is needed and `spec_helper.rb` prompts for its path in line 20. The only functionality specific to Dilithium keys is the safe destruction of key pointer values with `PrivateKey::Base::DILITHIUM.cleanup`, which is checked by the test case in line 100 of `specs/private_key/dilithium_spec.rb`. All test cases can be run by `bundle exec rspec -f d`. Ultimately, the adapted `README.md` includes a brief paragraph about Dilithium and usage examples for the added functionality.

This section is concluded by a brief demonstration of both migration results' interplay. A signature for the commit data in Source Code 23 from the git repository [Sch24a] can now be verified using `ssh_data`. This is demonstrated by executing the Ruby script in Source Code 24.

```
1  tree 45287fdcdfc6cb87a1d2fdd4afb3279f69a0dd16
2  parent 43e9cae2747742f09edaaf831ff7bb56f95f6b9d
3  author flck <florian.schroeck@stud.htwk-leipzig.de> 1705685521 +0100
4  committer flck <florian.schroeck@stud.htwk-leipzig.de> 1705685521
   ↪ +0100
5
6  Change README
```

Source Code 23. Commit data to be signed

```
1  require 'ssh_data'
2
3  # Extract commit data signed by git
4  commit_data = `git verify-commit HEAD -v 2> /dev/null`
5
6  # Extract SSH signature string produced by git
7  git_commit_string = `git cat-file commit HEAD`
8  start = git_commit_string.index("-----BEGIN SSH SIGNATURE-----")
9  stop = git_commit_string.index("-----END SSH SIGNATURE-----") + 27
10 signature_string = git_commit_string[start...stop]
11
12 # Create ssh_data Signature object
13 signature = SSHData::Signature.parse_pem(signature_string)
14
15 # Verify and print status
16 puts "Signature valid?: " + signature.verify(commit_data).to_s # true
```

Source Code 24. Verify Dilithium5 git commit signature with `ssh_data`

This thesis aimed to transfer theoretical research about PQC and PQC migrations into practical use by applying it to the source code of GitLab. Along the way, the following contribution was made:

Thesis contribution

- RubyCrypt
  - provided a Static Program Analysis tool for Ruby applications to support their Cryptographic Inventory Compilation ([Sch24b])
- Dilithium5 commit signatures in git
  - provided a proof of concept with working configuration ([Sch24a])
- Dilithium5 integration into `ssh_data`
  - enabled generation & parsing of Dilithium5 certificates and signatures in SSH encoding as well as certificate and signature verification using Ruby ([Sch24c])

As the results presented in section 4.1.1 reveal, RubyCrypt's applicability is restricted to the analysis of native Ruby applications that eagerly load their dependencies and avoid lazy loading functionalities. Although this a considerable limitation, such applications definitely exist. The following analysis supports this claim: by adapting RubyCrypt to only search for `autoload` commands in the source code of GitLab's 506 installed gems, the author determined that only 64 of these gems make use of autoloading. RubyCrypt could therefore still be used to support a CIC of the other 442 gems themselves.

Some adjustments to mitigate RubyCrypt's observed shortcomings could increase its utility. To avoid the parsing errors, the corresponding files could be transcoded to mask non UTF-8 characters, which can almost certainly be considered cryptographically irrelevant. To retrieve a loading target, even in the case of further program logic in the loading command, execution of the logic or more sophisticated scanning rules could be attempted. The latter could be fostered by employing dedicated tools for rule definition, such as RuboCop AST ([Bat24]). A methodologically different approach would be to search for class, module and method names defined in crypto gems instead of Ruby loading commands. However, this approach could not locate code that depends transitively on crypto gems.

Apart from further work on RubyCrypt, alternative SAST/SCA solutions supporting the CIC could be evaluated. SemGrep ([Inc24g]) is a versatile SAST tool whose rule set could be complemented by custom rules to locate API calls to crypto gems. During the period of writing, GitHub published its CI/CD component cbom ([Inc23]), claiming to automatically generate a Cryptography Bill of Materials. Last but not least, SecObserve ([Gmb23]) as an Open Source Crypto Agility Suite has recently been released. It

additionally uses Dynamic Application Security Testing (DAST) tools to investigate applications at runtime, which promises interesting results.

To complete the migration of GitLab's commit verification feature in particular, the remaining tasks are the integration of `ssh_data` into GitLab as well as exhaustive integration testing. With the preparatory work executed throughout this thesis, the necessary effort is reduced drastically. The inclusion of *all* PQC signature schemes provided by liboqs (i.e. all variants of Dilithium, but also Falcon and SPHINCS+) into `ssh_data` offers a possibility for future work and is a mere matter of analogously repeating the steps made for Dilithium5. The inclusion of hybrid signatures could also be realized in `ssh_data`, though it is still reliant on a consolidation of the recommendations for combiners.

It is to be expected that projects such as AMiQuaSy will produce further best practices supporting the numerous PQC migrations to come. Moreover, the upcoming publication of final Federal Information Processing Standards (FIPS) for the chosen algorithms of the NIST PQC standardization are likely to further increase the industry's attention towards PQC migrations. As regards the Open Source community, its willingness to further harden PQC libraries such as liboqs for a use in productive environments is shown by the launch of the Post-Quantum Cryptography Alliance in February 2024. The alliance includes projects such as Open Quantum Safe with a permanent funding by The Linux Foundation (see [Fou24a]). Cryptographic Abstraction APIs such as Sandwich ([San24b] or eUcrite ([ZH20]), aiming to hide the use of a particular cryptographic library behind a unified interface, attempt to meet the high need for cryptoagility by fostering a swift exchange of underlying primitives. This will be crucial to maintain security in the surprisingly fast world of modern cryptography.

6. References

[AD17] Martin R. Albrecht; Amit Deo: *Large Modulus Ring-LWE >= Module-LWE*. 2017. URL: https://eprint.iacr.org/2017/612 (visited on 02/16/2024). preprint

[Ajt96] 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. New York, NY, USA: Association for Computing Machinery, July 1, 1996, pp. 99–108. DOI: 10.1145/237814.237838. URL: https://dl.acm.org/doi/10.1145/237814.237838 (visited on 02/15/2024)

[Alb+23] Martin R. Albrecht et al.: "Variational Quantum Solutions to the Shortest Vector Problem." In: *Quantum* 7 (Mar. 2, 2023), p. 933. DOI: 10.22331/q-2023-03-02-933. arXiv: 2202.06757 [quant-ph]. URL: http://arxiv.org/abs/2202.06757 (visited on 02/16/2024)

[Aln+23] Nouri Alnahawi et al.: *On the State of Crypto-Agility*. 2023. URL: https://eprint.iacr.org/2023/487 (visited on 11/01/2023). preprint

[ANS22] ANSSI - French National Agency for the Security of Information Systems: *ANSSI Views on the Post-Quantum Cryptography Transition*. Mar. 30, 2022. URL: https://www.ssi.gouv.fr/publication/anssi-views-on-the-post-quantum-cryptography-transition/ (visited on 11/02/2023)

[Ark+24] André Arko et al.: *Bundler*. Version 2.5.5. Jan. 18, 2024. URL: https://bundler.io/ (visited on 01/26/2024)

[Att+23] Thomas Attema et al.: *PQC Migration Handbook*. Ed. by TNO - Netherlands Organisation for Applied Scientific Research; CWI - Netherlands National Research Institute for Mathematics and Computer Science; AIVD - Netherlands National Communications Security Agency. Mar. 2023. URL: https://www.tno.nl/en/newsroom/2023/04-0/pqc-migration-handbook/ (visited on 06/23/2023)

[Bat24] Bozhidar Batsov: *RuboCop AST*. 2024. URL: https://docs.rubocop.org/rubocop-ast/ (visited on 01/30/2024)

[BBD09] Daniel J. Bernstein; Johannes Buchmann; Erik Dahmén: *Post-Quantum Cryptography*. Berlin: Springer, 2009. 245 pp.

[Bec23] Jared Beck: *Bundler-Why*. Dec. 2, 2023. URL: https://github.com/jaredbeck/bundler-why (visited on 01/29/2024)

[Ber+22a] Daniel J. Bernstein et al.: *Classic McEliece: Conservative Code-Based Cryptography: Guide for Security Reviewers*. Oct. 23, 2022. URL: https:

//classic.mceliece.org/mceliece-security-20221023.pdf (visited on 07/04/2023)

[Ber+22b] Daniel J. Bernstein et al.: *Post-Quantum Cryptography: Integration Study*. Ed. by ENISA - European Union Agency for Cybersecurity. Publications Office of the European Union, 2022. URL: https://data.europa.eu/doi/10.2824/151162 (visited on 11/02/2023)

[Ber+23] Daniel J. Bernstein et al.: *Classic McEliece: Intro*. Mar. 4, 2023. URL: https://classic.mceliece.org/ (visited on 07/04/2023)

[Ber15] Elwyn R. Berlekamp: *Algebraic Coding Theory (Revised Edition)*. World Scientific, Mar. 26, 2015. 501 pp.

[Ber73] Elwyn R. Berlekamp: "Goppa Codes." In: *IEEE Transactions on Information Theory* IT-19.5 (Sept. 5, 1973), pp. 590–591. URL: https://web.archive.org/web/20170829142555/http://infosec.seu.edu.cn/space/kangwei/senior_thesis/Goppa.pdf (visited on 09/09/2023)

[Beu+21] Ward Beullens et al.: *Post-Quantum Cryptography: Current State and Quantum Mitigation*. Ed. by ENISA - European Union Agency for Cybersecurity. ENISA, Feb. 9, 2021. DOI: 10.2824/92307. URL: https://doi.org/10.2824/92307

[BH23] Nina Bindel; Britta Hale: *A Note on Hybrid Signature Schemes*. 2023. URL: https://eprint.iacr.org/2023/423 (visited on 02/08/2024). preprint

[Big08] Norman Biggs: *Codes: An Introduction to Information Communication and Cryptography*. Springer Undergraduate Mathematics Series. London: Springer, 2008

[Bin+17] Nina Bindel et al.: *Transitioning to a Quantum-Resistant Public Key Infrastructure*. 2017. URL: https://eprint.iacr.org/2017/460 (visited on 10/31/2023). preprint

[Bin+19] Nina Bindel et al.: "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange." In: *Post-Quantum Cryptography*. Ed. by Jintai Ding; Rainer Steinwandt. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 206–226. DOI: 10.1007/978-3-030-25510-7_12

[BL19] Daniel J. Bernstein; Tanja Lange: *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. Aug. 5, 2019. URL: https://bench.cr.yp.to/index.html (visited on 01/15/2024)

[BLS] Daniel J. Bernstein; Tanja Lange; Peter Schwabe: *NaCl: Networking and Cryptography Library*. URL: https://nacl.cr.yp.to/index.html (visited on 11/10/2023)

[BMvT78] E. Berlekamp; R. McEliece; H. van Tilborg: "On the Inherent Intractability of Certain Coding Problems (Corresp.)" In: *IEEE Transactions on Information Theory* 24.3 (May 1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873

[Bos+15] Joppe W. Bos et al.: "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem." In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. May 2015, pp. 553–570. DOI: 10.1109/SP.2015.40. URL: https://ieeexplore.ieee.org/document/7163047 (visited on 10/24/2023)

[Bra+19] Alexandre Braga et al.: "Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software." In: *IEEE Transactions on Reliability* 68.4 (Dec. 2019), pp. 1384–1403. DOI: 10.1109/TR.2019.2937214. URL: https://ieeexplore.ieee.org/document/8863426/ (visited on 01/22/2024)

[BSN21] William Barker; Murugiah Souppaya; William Newhouse: *Migration to Post-Quantum Cryptography*. Aug. 2021. URL: https://www.nccoe.nist.gov/sites/default/files/2022-07/pqc-migration-project-description-final.pdf

[BSW21] Jenny Blessing; Michael A. Specter; Daniel J. Weitzner: *You Really Shouldn't Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries*. July 10, 2021. DOI: 10.48550/arXiv.2107.04940. arXiv: 2107.04940 [cs]. URL: http://arxiv.org/abs/2107.04940 (visited on 11/09/2023). preprint

[Buc+17] Johannes Buchmann et al.: *Evaluation of Lattice-Based Cryptographic Algorithms*. Ed. by Bundesamt für Sicherheit in der Informationstechnik. 2017. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Gitterbasierte_Verfahren/Gitterbasierte_Verfahren.pdf?__blob=publicationFile&v=1 (visited on 07/27/2023)

[Bun+24] Bundesamt für Sicherheit in der Informationstechnik et al.: *Position Paper on Quantum Key Distribution*. Jan. 24, 2024. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Quantum_Positionspapier.html?nn=916626

[Bun20] Bundesamt für Sicherheit in der Informationstechnik: *Migration to Post Quantum Cryptography - Recommendations for action by the BSI*. Aug. 2020. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/

BSI/Crypto/Migration_to_Post_Quantum_Cryptography.pdf?__blob=publicationFile&v=2

[Bun21] Bundesamt für Sicherheit in der Informationstechnik: *Quantum-Safe Cryptography – Fundamentals, Current Developments and Recommendations*. Oct. 2021. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Brochure/quantum-safe-cryptography.pdf?__blob=publicationFile&v=6 (visited on 11/06/2023)

[CH24] The MITRE Corporation; US Department of Homeland Security (DHS) Cybersecurity and Infrastructure Security Agency: *Common Vulnerabilities and Exposures (CVE) Program*. 2024. URL: https://www.cve.org/ (visited on 01/23/2024)

[Cis20] Inc. Cisco Systems: "Configuring Post-Quantum MACsec in Cisco Switches." In: (2020). URL: https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/configuring-post-quantum-macsec-in-cisco-switches.pdf

[Com] OpenSSL Management Comittee: *OpenSSL*. URL: https://www.openssl.org/ (visited on 11/10/2023)

[Com23] Atom Computing: *Quantum Startup Atom Computing First to Exceed 1,000 Qubits*. Atom Computing. Oct. 24, 2023. URL: https://atom-computing.com/quantum-startup-atom-computing-first-to-exceed-1000-qubits/ (visited on 02/11/2024)

[Cor23a] IBM Corporation: *CBOM - Cryptography Bill of Materials*. Version 1.4-cbom-1.0. IBM Corporation, June 29, 2023. URL: https://github.com/IBM/CBOM (visited on 01/24/2024)

[Cor23b] IBM Corporation: *IBM Quantum Safe Technology*. 2023. URL: https://ibm-com-qc-dev.quantum-computing.ibm.com/quantum/assets/quantum-safe/IBM_Quantum-Safe-Technology-Brochure.pdf (visited on 01/23/2024)

[Cor23c] IBM Corporation: *IBM Quantum System Two: The Era of Quantum Utility Is Here*. IBM Research Blog. Dec. 4, 2023. URL: https://research.ibm.com/blog/quantum-roadmap-2033 (visited on 02/11/2024)

[CP05] Richard E. Crandall; Carl Pomerance: *Prime Numbers: A Computational Persprective*. 2nd ed. New York: Springer, 2005

[CPS19] Eric Crockett; Christian Paquin; Douglas Stebila: "Prototyping Post-Quantum and Hybrid Key Exchange and Authentication in TLS and SSH." In: *Cryptology ePrint Archive* (2019). URL: https://eprint.iacr.org//2019/858 (visited on 09/27/2023)

[CS03] Ronald Cramer; Victor Shoup: "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack." In: *SIAM Journal on Computing* 33.1 (Jan. 2003), pp. 167–226. DOI: 10.1137/S0097539702403773. URL: http://epubs.siam.org/doi/10.1137/S0097539702403773 (visited on 08/17/2023)

[CS21] Joo Yeon Cho; Andrew Sergeev: "TLV-to-MUC Express: Post-quantum MACsec in VXLAN." In: *Secure IT Systems*. Ed. by Mikael Asplund; Simin Nadjm-Tehrani. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 127–141. DOI: 10.1007/978-3-030-70852-8_8

[CS23] Scott Chacon; Ben Straub: *Pro Git Book*. Dec. 22, 2023. URL: https://git-scm.com/book/en/v2 (visited on 01/20/2024)

[CW07] Brian Chess; Jacob West: *Secure Programming with Static Analysis*. Addison-Wesley Professional, June 2007. URL: https://learning.oreilly.com/library/view/secure-programming-with/9780321424778/ (visited on 01/22/2024)

[De 17] Luca De Feo: *Mathematics of Isogeny Based Cryptography*. Nov. 10, 2017. arXiv: 1711.04062 [cs, math]. URL: http://arxiv.org/abs/1711.04062 (visited on 02/16/2024). preprint

[DH76] W. Diffie; M. Hellman: "New Directions in Cryptography." In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638

[Ege+13] Manuel Egele et al.: "An Empirical Study of Cryptographic Misuse in Android Applications." In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. The 2013 ACM SIGSAC Conference. Berlin, Germany: ACM Press, 2013, pp. 73–84. DOI: 10.1145/2508859.2516693. URL: http://dl.acm.org/citation.cfm?doid=2508859.2516693 (visited on 10/02/2023)

[EOS07] D. Engelbert; R. Overbeck; A. Schmidt: "A Summary of McEliece-Type Cryptosystems and Their Security." In: *Journal of Mathematical Cryptology* 1.2 (Apr. 1, 2007), pp. 151–199. DOI: 10.1515/JMC.2007.009. URL: https://www.degruyter.com/document/doi/10.1515/JMC.2007.009/html (visited on 07/04/2023)

[Fou] The Rails Foundation: *Autoloading and Reloading Constants — Ruby on Rails Guides v7.1.3*. URL: https://guides.rubyonrails.org/v7.1.3/autoloading_and_reloading_constants.html (visited on 01/31/2024)

[Fou+] Pierre-Alain Fouque et al.: *Falcon - Fast-Fourier Lattice-based Compact Signatures over NTRU*. URL: https://falcon-sign.info/ (visited on 02/03/2024)

[Fou24a] The Linux Foundation: *Post-Quantum Cryptography Alliance Launches to Advance Post-Quantum Cryptography*. Feb. 6, 2024. URL: https://www.linuxfoundation.org/press/announcing-the-post-quantum-cryptography-alliance-pqca (visited on 02/07/2024)

[Fou24b] The Rails Foundation: *Ruby on Rails*. Ruby on Rails. Jan. 16, 2024. URL: https://rubyonrails.org/ (visited on 01/31/2024)

[Fre22] Charlotte Freeman: *Five Cryptography Best Practices for Developers | Synopsys Blog*. Jan. 18, 2022. URL: https://www.synopsys.com/blogs/software-security/cryptography-best-practices.html (visited on 11/09/2023)

[Gaz+23] Stefan-Lukas Gazdag et al.: "Quantum-Resistant MACsec and IPsec for Virtual Private Networks." In: *Security Standardisation Research*. Ed. by Felix Günther; Julia Hesse. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 1–21. DOI: 10.1007/978-3-031-30731-7_1

[gCod24] g10 Code GmbH: *The GNU Privacy Guard (GnuPG)*. Jan. 25, 2024. URL: https://gnupg.org/index.html (visited on 02/03/2024)

[GE21] Craig Gidney; Martin Ekerå: "How to Factor 2048 Bit RSA Integers in 8 Hours Using 20 Million Noisy Qubits." In: *Quantum* 5 (Apr. 15, 2021), p. 433. DOI: 10.22331/q-2021-04-15-433. URL: https://quantum-journal.org/papers/q-2021-04-15-433/ (visited on 02/11/2024)

[GgGA23] XITASO GmbH; genua GmbH; Ostbayerische Technische Hochschule Amberg-Weiden: *AMiQuaSy — Vernetzung Und Sicherheit Digitaler Systeme*. 2023. URL: https://www.forschung-it-sicherheit-kommunikationssysteme.de/projekte/amiquasy (visited on 10/25/2023)

[git24] git: *Git Source Code*. Git, Feb. 5, 2024. URL: https://github.com/git/git (visited on 02/05/2024)

[Glo23] InfoSec Global: *AgileSec Analytics*. 2023. URL: https://www.infosecglobal.com/solutions/agilesec-analytics (visited on 01/23/2024)

[Gmb23] MaibornWolff GmbH: *SecObserve*. Version 1.4.0. 2023. URL: https://github.com/MaibornWolff/SecObserve/tree/main (visited on 01/23/2024)

[Gop70] Valerii Denisovich Goppa: "A New Class of Linear Correcting Codes." In: *Problemy Peredachi Informatsii* 6.3 (1970), pp. 24–30. URL: https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=1748&option_lang=eng (visited on 09/10/2023)

[Gop71] Valerii Denisovich Goppa: "Rational Representation of Codes and (L, g)-Codes." In: *Problemy Peredachi Informatsii* 7.3 (1971), pp. 41–49. URL: https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=1647&option_lang=eng (visited on 09/10/2023)

[Gop72] Valerii Denisovich Goppa: "Codes Constructed on the Base of (L,g)-Codes." In: *Problemy Peredachi Informatsii* 8.2 (1972), pp. 107–109. URL: https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=ppi&paperid=791&option_lang=eng (visited on 09/10/2023)

[Gro96] Lov K. Grover: *A Fast Quantum Mechanical Algorithm for Database Search*. Nov. 19, 1996. DOI: 10.48550/arXiv.quant-ph/9605043. arXiv: quant-ph/9605043. URL: http://arxiv.org/abs/quant-ph/9605043 (visited on 06/21/2023). preprint

[GS15] Anjana Gosain; Ganga Sharma: "Static Analysis: A Survey of Techniques and Tools." In: *Intelligent Computing and Applications*. Ed. by Durbadal Mandal et al. Advances in Intelligent Systems and Computing. New Delhi: Springer India, 2015, pp. 581–591. DOI: 10.1007/978-81-322-2268-2_59

[Hei19] Tobias Heider: *Towards a Verifiably Secure Quantum-Resistant Key Exchange in IKEv2*. Oct. 28, 2019

[Hek+23] Julius Hekkala et al.: "Implementing Post-quantum Cryptography for Developers." In: *SN Computer Science* 4.4 (Apr. 29, 2023), p. 365. DOI: 10.1007/s42979-023-01724-1. URL: https://doi.org/10.1007/s42979-023-01724-1 (visited on 11/09/2023)

[HGL21] Daniel Herzinger; Stefan-Lukas Gazdag; Daniel Loebenberger: "Real-World Quantum-Resistant IPsec." In: *2021 14th International Conference on Security of Information and Networks (SIN)*. 2021 14th International Conference on Security of Information and Networks (SIN). Vol. 1. Dec. 2021, pp. 1–8. DOI: 10.1109/SIN54109.2021.9699255. URL: https://ieeexplore.ieee.org/document/9699255/citations?tabFilter=papers#citations (visited on 10/24/2023)

[Hou15] Russ Housley: *Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms*. Request for Comments RFC 7696. Internet Engineering Task Force, Nov. 2015. 19 pp. DOI: 10.17487/RFC7696. URL: https://datatracker.ietf.org/doc/rfc7696 (visited on 01/15/2024)

[Hou22] The White House: *National Security Memorandum on Promoting United States Leadership in Quantum Computing While Mitigating Risks to Vulnerable Cryptographic Systems*. May 4, 2022. URL: https://www.whitehouse.gov/briefing-room/statements-releases/2022/05/04/national-security-memorandum-on-promoting-united-states-leadership-in-quantum-computing-while-mitigating-risks-to-vulnerable-cryptographic-systems/ (visited on 11/06/2023)

[HPS98] Jeffrey Hoffstein; Jill Pipher; Joseph H. Silverman: "NTRU: A Ring-Based Public Key Cryptosystem." In: *Algorithmic Number Theory*. Ed. by Joe P. Buhler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 267–288. DOI: 10.1007/BFb0054868

[Hül+] Andreas Hülsing et al.: *SPHINCS+*. URL: https://sphincs.org/index.html (visited on 02/03/2024)

[Inc20] Ribose Inc.: *Ruby-Botan*. Feb. 4, 2020. URL: https://github.com/riboseinc/ruby-botan (visited on 02/07/2024)

[Inc22] Stack Exchange Inc.: *Stack Overflow Developer Survey 2022 - Version Control Platform*. Stack Overflow. 2022. URL: https://survey.stackoverflow.co/2022/#section-version-control-version-control-platforms (visited on 01/20/2024)

[Inc23] GitHub Inc.: *Cbom-Action*. Version 1.0.1. Nov. 28, 2023. URL: https://github.com/advanced-security/cbom-action/releases (visited on 01/24/2024)

[Inc24a] GitHub Inc.: *GitHub Docs*. GitHub Docs. 2024. URL: https://docs.github.com/_next/data/QXyaA0Ck2gpL2rUMLUhsW/en.json (visited on 02/03/2024)

[Inc24b] GitHub Inc.: *Ssh_data*. Version 1.3.0. GitHub, Feb. 1, 2024. URL: https://github.com/github/ssh_data (visited on 02/03/2024)

[Inc24c] GitLab Inc.: *About GitLab*. Jan. 15, 2024. URL: https://about.gitlab.com/ (visited on 01/15/2024)

[Inc24d] GitLab Inc.: *About GitLab - Company*. 2024. URL: https://about.gitlab.com/company (visited on 01/18/2024)

[Inc24e] GitLab Inc.: *GitLab Docs*. 2024. URL: https://docs.gitlab.com/ee/ (visited on 01/20/2024)

[Inc24f] GitLab Inc.: *GitLab Git Repository*. GitLab. Jan. 16, 2024. URL: https://gitlab.com/gitlab-org/gitlab-foss (visited on 01/16/2024)

[Inc24g] Semgrep Inc.: *Semgrep Code*. Semgrep Code: Deploy, manage, and monitor Semgrep at scale. 2024. URL: https://semgrep.dev/products/semgrep-code/ (visited on 01/23/2024)

[Inc24h] Synopsys Inc.: *Black Duck SCA*. 2024. URL: https://www.synopsys.com/software-integrity/software-composition-analysis-tools.html (visited on 01/23/2024)

[Inf+20] FZI Forschungszentrum Informatik et al.: *QuantumLeap*. Nov. 2020. URL: https://projekt-quantumleap.de/ (visited on 10/29/2023)

[Ins20] European Telecommunications Standards Institute: *Migration Strategies and Recommendations to Quantum Safe Schemes*. Technical Report. Version 1.1.1. July 21, 2020. URL: https://www.etsi.org/deliver/etsi_tr/103600_103699/103619/01.01.01_60/tr_103619v010101p.pdf (visited on 06/21/2023)

[Ins22] European Telecommunications Standards Institute: *Cyber Security (CYBER); Critical Security Controls for Effective Cyber Defence; Part 1: The Critical Security Controls*. Technical Report. Version 4.1.2. Apr. 14, 2022. URL: https://www.etsi.org/deliver/etsi_tr/103300_103399/10330501/04.01.02_60/tr_10330501v040102p.pdf (visited on 10/27/2023)

[Ins23] European Telecommunications Standards Institute: *Quantum-Safe Cryptography (QSC) Migration; ITS and C-ITS Migration Study*. Technical Report. Version 1.1.1. May 12, 2023. URL: https://www.etsi.org/deliver/etsi_tr/103900_103999/103949/01.01.01_60/tr_103949v010101p.pdf (visited on 10/27/2023)

[Kan+22] Matthias J. Kannwischer et al.: *Improving Software Quality in Cryptography Standardization Projects*. 2022. URL: https://eprint.iacr.org/2022/337 (visited on 01/15/2024). preprint

[Kan+23] Matthias J. Kannwischer et al.: *PQM4: Post-quantum Crypto Library for the ARM Cortex-M4*. mupq, Oct. 25, 2023. URL: https://github.com/mupq/pqm4 (visited on 10/25/2023)

[KB23] KPMG AG; Bundesamt für Sicherheit in der Informationstechnik: *Market Survey on Cryptography and Quantum Computing*. 2023. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Crypto/Marktumfrage_EN_Kryptografie_Quantencomputing.pdf?__blob=publicationFile&v=3 (visited on 11/06/2023)

[Kob87] Neal Koblitz: "Elliptic Curve Cryptosystems." In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. DOI: 10.2307/2007884. JSTOR: 2007884. URL: https://www.jstor.org/stable/2007884 (visited on 02/12/2024)

[Krü+17] Stefan Krüger et al.: "CogniCrypt: Supporting Developers in Using Cryptography." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Oct. 2017, pp. 931–936. DOI: 10.1109/ASE.2017.8115707. URL: https://ieeexplore.ieee.org/document/8115707 (visited on 01/23/2024)

[KSH23] Panos Kampanakis; Douglas Stebila; Torben Hansen: *Post-Quantum Hybrid Key Exchange in SSH*. Internet Draft - Work in Progress. Apr. 10, 2023. URL: https://datatracker.ietf.org/doc/draft-kampanakis-curdle-ssh-pq-ke (visited on 11/03/2023)

[KSK09] Uday Khedker; Amitabha Sanyal; Bageshri Karkare: *Data Flow Analysis: Theory and Practice*. Boca Raton, FL: CRC Press/Taylor & Francis, 2009

[KY21] Jonathan Katz; Lindell Yehuda: *Introduction to Modern Cryptography*. Third edition. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press Taylor & Francis Group, 2021

[Leg] Legion of the Bouncy Castle Inc.: *Bouncy Castle*. URL: https://www.bouncycastle.org/index.html (visited on 11/10/2023)

[Li+22] Wenqing Li et al.: "CryptoGo: Automatic Detection of Go Cryptographic API Misuses." In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. New York, NY, USA: Association for Computing Machinery, Dec. 5, 2022, pp. 318–331. DOI: 10.1145/3564625.3567989. URL: https://dl.acm.org/doi/10.1145/3564625.3567989 (visited on 10/02/2023)

[Lia23] Chris Liaw: *Roqs*. Version 0.1.1. Nov. 24, 2023. URL: https://github.com/chrisliaw/roqs/blob/0.1.1/roqs.gemspec (visited on 02/07/2024)

[Lim24] Micro Focus International Limited: *Fortify Static Code Analyzer*. 2024. URL: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer (visited on 01/23/2024)

[Liu+22] Jie Liu et al.: "An Improved McEliece Cryptosystem Based on QC-MDPC Code with Compact Key Size." In: *Telecommunication Systems* 80.1 (May 1, 2022), pp. 17–32. DOI: 10.1007/s11235-022-00881-7. URL: https://doi.org/10.1007/s11235-022-00881-7 (visited on 02/16/2024)

[LL93] Arjen K. Lenstra; Hendrik W. Lenstra, eds.: *The Development of the Number Field Sieve*. Vol. 1554. Lecture Notes in Mathematics. Berlin, Heidelberg: Springer, 1993. DOI: 10.1007/BFb0091534. URL: http://link.springer.com/10.1007/BFb0091534 (visited on 02/11/2024)

[LLL82] A. K. Lenstra; H. W. Lenstra; L. Lovász: "Factoring Polynomials with Rational Coefficients." In: *Mathematische Annalen* 261.4 (Dec. 1, 1982), pp. 515–534. DOI: 10.1007/BF01457454. URL: https://doi.org/10.1007/BF01457454 (visited on 02/15/2024)

[Llo+23] Jack Lloyd et al.: *Botan: Crypto and TLS for Modern C++*. Oct. 26, 2023. URL: https://github.com/randombit/botan (visited on 10/29/2023)

[LPR10] Vadim Lyubashevsky; Chris Peikert; Oded Regev: "On Ideal Lattices and Learning with Errors over Rings." In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 1–23. DOI: 10.1007/978-3-642-13190-5_1

[LY06a] Chris M. Lonvick; Tatu Ylonen: *The Secure Shell (SSH) Authentication Protocol*. Request for Comments RFC 4252. Internet Engineering Task Force, Jan. 2006. 17 pp. DOI: 10.17487/RFC4252. URL: https://datatracker.ietf.org/doc/rfc4252 (visited on 01/20/2024)

[LY06b] Chris M. Lonvick; Tatu Ylonen: *The Secure Shell (SSH) Protocol Architecture*. Request for Comments RFC 4251. Internet Engineering Task Force, Jan. 2006. 30 pp. DOI: 10.17487/RFC4251. URL: https://datatracker.ietf.org/doc/rfc4251 (visited on 02/05/2024)

[LY06c] Chris M. Lonvick; Tatu Ylonen: *The Secure Shell (SSH) Transport Layer Protocol*. Request for Comments RFC 4253. Internet Engineering Task Force, Jan. 2006. 32 pp. DOI: 10.17487/RFC4253. URL: https://datatracker.ietf.org/doc/rfc4253 (visited on 02/05/2024)

[Lyu+] Vadim Lyubashevsky et al.: *CRYSTALS-Dilithium*. URL: https://pq-crystals.org/dilithium/index.shtml (visited on 02/03/2024)

[Lyu+21] Vadim Lyubashevsky et al.: *CRYSTALS-Dilithium - Algorithm Specifications and Supporting Documentation (Version 3.1)*. Feb. 8, 2021. URL: https://pq-crystals.org/dilithium/index.shtml (visited on 02/03/2024)

[McE78] Robert J. McEliece: "A Public-Key Cryptosystem Based On Algebraic Coding Theory." In: *The Deep Space Network Progress Report 42-44*. Apr. 15, 1978, pp. 114–116. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (visited on 07/04/2023)

[MH] Tyson Macaulay; Richard Henderson: *Cryptographic Agility in Practice*. URL: https://assets.website-files.com/5bd73d456f7b3f2db2bbbb95/5c76a740dcc2cc4646a06805_ISG_AgilityUseCases_Whitepaper-FINAL.pdf (visited on 09/29/2023)

[Mil86] Victor S. Miller: "Use of Elliptic Curves in Cryptography." In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1986, pp. 417–426. DOI: 10.1007/3-540-39799-X_31

[Mos15] Michele Mosca: *Cybersecurity in an Era with Quantum Computers: Will We Be Ready?* 2015. URL: https://eprint.iacr.org/2015/1075 (visited on 09/24/2023). preprint

[MP23] Michele Mosca; Marco Piani: *2023 Quantum Threat Timeline Report*. Global Risk Institute. Dec. 22, 2023. URL: https://globalriskinstitute.org/publication/2023-quantum-threat-timeline-report/ (visited on 02/12/2024)

[Nat17] Information Technology Laboratory National Institute of Standards and Technology: *Post-Quantum Cryptography Standardization*. CSRC | NIST. Jan. 3, 2017. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization (visited on 10/25/2023)

[Nat20] National Cyber Security Center: *Preparing for Quantum-Safe Cryptography*. Nov. 11, 2020. URL: https://www.ncsc.gov.uk/whitepaper/preparing-for-quantum-safe-cryptography

[NAT24] North Atlantic Treaty Organization (NATO): *Summary of NATO's Quantum Technologies Strategy*. NATO. Jan. 17, 2024. URL: https://www.nato.int/cps/en/natohq/official_texts_221777.htm (visited on 01/30/2024)

[Nel19] Nels Nelson: *bouncycastle*. GitLab. Sept. 30, 2019. URL: https://gitlab.com/nelsnelson/bouncycastle (visited on 02/07/2024)

[Nie86] Harald Niederreiter: "Knapsack-Type Cryptosystems and Algebraic Coding Theory." In: *Problems of Control and Information Theory* 15.2 (1986), pp. 159–166

[Ols24] Christoph Olsowka: *The Ruby Toolbox - Encryption & Cryptography*. The Ruby Toolbox. 2024. URL: https://www.ruby-toolbox.com/categories/encryption (visited on 01/25/2024)

[OPp19] David Ott; Christopher Peikert; other workshop participants: *Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility*. Sept. 16, 2019. DOI: 10.48550/arXiv.1909.07353. arXiv: 1909.07353 [cs]. URL: http://arxiv.org/abs/1909.07353 (visited on 09/28/2023). preprint

[Par] Aaron Parecki: *OAuth 2.0 — OAuth*. URL: https://oauth.net/2/ (visited on 02/02/2024)

[Pat75] N. Patterson: "The Algebraic Decoding of Goppa Codes." In: *IEEE Transactions on Information Theory* 21.2 (Mar. 1975), pp. 203–207. DOI: 10.1109/TIT.1975.1055350

[Pau22] Sebastian Paul: "On the Transition to Post-Quantum Cryptography in the Industrial Internet of Things." PhD thesis. Darmstadt: Technische Universität Darmstadt, 2022. DOI: 10.26083/tuprints-00021368. URL: https://tuprints.ulb.tu-darmstadt.de/21368/ (visited on 06/16/2023)

[Pos20] Howard Poston: *Cryptography-Based Vulnerabilities in Applications | InfoSec Institute Blog*. July 29, 2020. URL: https://resources.infosecinstitute.com/topics/secure-coding/cryptography-based-vulnerabilities-in-applications/ (visited on 11/09/2023)

[Pro] OpenBSD Project: *OpenSSH*. URL: https://www.openssh.com/ (visited on 11/10/2023)

[Pro23] OpenBSD Project: *OpenSSH: Release Notes*. Oct. 5, 2023. URL: https://www.openssh.com/releasenotes.html (visited on 10/25/2023)

[Qua+24] Nick Quaranto et al.: *RubyGems.Org*. 2024. URL: https://rubygems.org/ (visited on 01/26/2024)

[Reg05] Oded Regev: "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography." In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '05. New York, NY, USA: Association for Computing Machinery, May 22, 2005, pp. 84–93. DOI: 10.1145/1060590.1060603. URL: https://doi.org/10.1145/1060590.1060603 (visited on 02/15/2024)

[Reg09] Oded Regev: "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography." In: *Journal of the ACM* 56.6 (Sept. 2009), 34–34:40. DOI: 10.1145/1568318.1568324. URL: https://search.ebscohost.com/login.aspx?direct=true&db=buh&AN=44295900&lang=de&site=ehost-live (visited on 08/14/2023)

[Res18] Eric Rescorla: *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments RFC 8446. Internet Engineering Task Force, Aug. 2018. 160 pp. DOI: 10.17487/RFC8446. URL: https://datatracker.ietf.org/doc/rfc8446 (visited on 01/20/2024)

[Rot06] Ron M. Roth: *Introduction to Coding Theory*. Cambridge, UK ; New York: Cambridge University Press, 2006. 566 pp.

[Rub24] RubySec: *Ruby Advisory Database*. RubySec, Jan. 12, 2024. URL: https://github.com/rubysec/ruby-advisory-db (visited on 01/23/2024)

[Saf] Open Quantum Safe: *Liboqs*. URL: https://github.com/open-quantum-safe/liboqs (visited on 10/25/2023)

[Saf23a] Open Quantum Safe: *OpenSSH - Open Quantum Safe Fork*. Open Quantum Safe, Oct. 22, 2023. URL: https://github.com/open-quantum-safe/openssh (visited on 10/24/2023)

[Saf23b] Open Quantum Safe: *Oqs-Provider - Open Quantum Safe Fork of OpenSSL 3*. Open Quantum Safe, Oct. 22, 2023. URL: https://github.com/open-quantum-safe/oqs-provider/ (visited on 10/25/2023)

[San24a] SandboxAQ: *SandboxAQ Security Suite*. 2024. URL: https://www.sandboxaq.com/solutions/security-suite (visited on 01/23/2024)

[San24b] SandboxAQ: *Sandwich*. Version v0.1.0. SandboxAG, Jan. 8, 2024. URL: https://github.com/sandbox-quantum/sandwich (visited on 01/23/2024)

[Sch24a] Florian Schröck: *Git_with_pqc_signature*. Feb. 8, 2024. URL: https://github.com/FS93/git_with_pqc_signature (visited on 02/09/2024)

[Sch24b] Florian Schröck: *RubyCrypt*. GitHub. 2024. URL: https://github.com/FS93/RubyCrypt (visited on 01/26/2024)

[Sch24c] Florian Schröck: *Ssh_data Fork*. 2024. URL: https://github.com/FS93/ssh_data/tree/include-post-quantum-crypto (visited on 02/13/2024)

[Sch87] C. P. Schnorr: "A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms." In: *Theoretical Computer Science* 53.2 (Jan. 1, 1987), pp. 201–224. DOI: 10.1016/0304-3975(87)90064-8. URL: https://www.sciencedirect.com/science/article/pii/0304397587900648 (visited on 02/15/2024)

[SE] National Institute of Standards and Technology; National Cybersecurity Center of Excellence: *Migration to Post-Quantum Cryptography | NCCoE*. URL: https://www.nccoe.nist.gov/crypto-agility-considerations-migrating-post-quantum-cryptographic-algorithms (visited on 09/22/2023)

[SE94] C. P. Schnorr; M. Euchner: "Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems." In: *Mathematical Programming* 66.1 (Aug. 1, 1994), pp. 181–199. DOI: 10.1007/BF01581144. URL: https://doi.org/10.1007/BF01581144 (visited on 02/15/2024)

[Sha71] Daniel Shanks: "Class Number, a Theory of Factorization, and Genera." In: *1969 Number Theory Institute*. Red. by Donald Lewis. Vol. 20. Proceedings

of Symposia in Pure Mathematics. American Mathematical Society, 1971, pp. 415–440. DOI: 10.1090/pspum/020. URL: https://www.ams.org/pspum/020 (visited on 02/11/2024)

[Shi+24] Hiroshi Shibata et al.: *Ruby Language Documentation*. 2024. URL: https://www.ruby-lang.org/en/ (visited on 01/26/2024)

[Sho94] Peter W. Shor: "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700

[Sic23a] Bundesamt für Sicherheit in der Informationstechnik: *Cyber Resilience Requirements for Manufacturers and Products - Part2: Software Bill of Materials (SBOM)*. Version 1.1. Nov. 28, 2023. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Technische-Richtlinien/TR-nach-Thema-sortiert/tr03183/tr-03183.html?nn=1091552 (visited on 01/24/2024)

[Sic23b] Bundesamt für Sicherheit in der Informationstechnik: *Status of Quantum Computer Development - Entwicklungsstand Quantencomputer*. Aug. 2023. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/Quantencomputer/Entwicklungstand_QC_V_2_0.html (visited on 11/21/2023)

[Sin20] Harshdeep Singh: *Code Based Cryptography: Classic McEliece*. Version 2. May 29, 2020. DOI: 10.48550/arXiv.1907.12754. arXiv: 1907.12754 [cs]. URL: http://arxiv.org/abs/1907.12754 (visited on 07/04/2023). preprint

[SIT+19] Fraunhofer SIT et al.: *QuantumRISC*. Sept. 2019. URL: https://www.quantumrisc.de/index_en.html (visited on 11/01/2023)

[SKD20] Dimitrios Sikeridis; Panos Kampanakis; Michael Devetsikiotis: "Assessing the Overhead of Post-Quantum Cryptography in TLS 1.3 and SSH." In: Nov. 1, 2020. DOI: 10.1145/3386367.3431305

[Sta19] National Institute of Standards and Technology: *Security Requirements for Cryptographic Modules*. Federal Information Processing Standard (FIPS). Mar. 22, 2019. DOI: 10.6028/NIST.FIPS.140-3. URL: https://csrc.nist.gov/pubs/fips/140-3/final (visited on 11/06/2023)

[SWZ16] John M. Schanck; William Whyte; Zhenfei Zhang: *Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) Version 1.2*. Internet Draft - Work in Progress draft-whyte-qsh-tls12-02. Internet Engineering Task Force, July 22, 2016. 19 pp. URL: https://datatracker.ietf.org/doc/draft-whyte-qsh-tls12 (visited on 10/24/2023)

[TR22] Simran Tinani; Joachim Rosenthal: "A Deterministic Algorithm for the Discrete Logarithm Problem in a Semigroup." In: *Journal of Mathematical Cryptology* 16.1 (Jan. 1, 2022), pp. 141–155. DOI: 10.1515/jmc-2021-0022. URL: https://www.degruyter.com/document/doi/10.1515/jmc-2021-0022/html (visited on 02/11/2024)

[Tur08] Franklyn Turbak: *Design Concepts in Programming Languages*. In collab. with David K. Gifford; Mark A. Sheldon; Franklyn Albin Turbak. Cambridge, Mass. [u.a: : MIT Press, 2008. xxiv+1322

[Uni19] International Telecommunication Union: *X.509 : Information Technology - Open Systems Interconnection - The Directory: Public-key and Attribute Certificate Frameworks*. Version 9.0. Oct. 2019. URL: https://www.itu.int/rec/T-REC-X.509/en (visited on 10/31/2023). In Force

[Vid23] Martin Vidner: *Code-Explorer*. Sept. 22, 2023. URL: https://github.com/mvidner/code-explorer (visited on 01/29/2024)

[Wei] Justin Weiss: *How Does Rails Handle Gems?* Justin Weiss. URL: https://www.justinweiss.com/articles/how-does-rails-handle-gems/ (visited on 02/01/2024)

[whi+24] whitequark et al.: *Parser*. Jan. 29, 2024. URL: https://github.com/whitequark/parser (visited on 01/29/2024)

[Why+17] William Whyte et al.: *Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) Version 1.3*. Internet Draft - Work in Progress draft-whyte-qsh-tls13-06. Internet Engineering Task Force, Oct. 3, 2017. 19 pp. URL: https://datatracker.ietf.org/doc/draft-whyte-qsh-tls13 (visited on 10/24/2023)

[Wie+21] Alexander Wiesmaier et al.: *On PQC Migration and Crypto-Agility*. June 17, 2021. DOI: 10.48550/arXiv.2106.09599. arXiv: 2106.09599 [cs]. URL: http://arxiv.org/abs/2106.09599 (visited on 09/27/2023). preprint

[Wil99] Wolfgang Willems: *Codierungstheorie*. Berlin ; De Gruyter, 1999. DOI: 10.1515/9783110806267

[WXY23] Xiaoyun Wang; Guangwu Xu; Yang Yu: "Lattice-Based Cryptography: A Survey." In: *Chinese Annals of Mathematics, Series B* 44.6 (Nov. 1, 2023), pp. 945–960. DOI: 10.1007/s11401-023-0053-6. URL: https://doi.org/10.1007/s11401-023-0053-6 (visited on 02/14/2024)

[xmi24] xmisao: *Encryption Gems*. 2024. URL: https://bestgems.org/categories/Encryption (visited on 01/25/2024)

[ZH20] Alexander Zeier; Andreas Heinemann: *eUCRITE 1.0 API*. 2020. URL: https://cspub.h-da.io/eucrite/ (visited on 01/23/2024)

List of Tables

TABLE 1 Comparison of Software Tools for Cryptographic Inventory Compilation 27
TABLE 2 Key and signature sizes of PQC Digital Signature Algorithms in bytes 41

List of Figures

FIGURE 1 Estimation of time and qubits needed for factorization and solving of DLP using different approaches on QCs 2
FIGURE 2 Experts' estimates of the likelihood of a QC breaking RSA-2048 in 24 hours 3
FIGURE 3 Simplified Dilitium scheme 14
FIGURE 4 Distribution of Programming Languages in GitLab's Source Code 18
FIGURE 5 GitLab High-Level Architecture Overview 20
FIGURE 6 Structure of output directory produced by RUBYCRYPT . . . 35
FIGURE 7 Dilithium5 performance benchmarking for different architectures 42
FIGURE 8 Falcon-1024 performance benchmarking for different architectures 43
FIGURE 9 SPHINCS+-SHA2-256s-simple performance benchmarking for different architectures 44

List of Source Code

Source Code 1 Install GitLab CE v16.8.0 19
Source Code 2 Count the number of Ruby files in the GitLab repository . . . 19
Source Code 3 Extract from GitLab's Gemfile 29
Source Code 4 Extract from GitLab's Gemfile.lock 29
Source Code 5 Execution of RubyCrypt 30
Source Code 6 Extract from a JSON file containing gem summaries 31
Source Code 7 Depth-first search to calculate dependency chains 32
Source Code 8 Extract from output file `dependency_chains.json` 33
Source Code 9 Filter ASTs for loading commands 34
Source Code 10 Extract from output file `requirements.json` 35
Source Code 11 Statistics about RubyCrypt results produced by `analysis.rb` 36
Source Code 12 Requirement with program logic 37
Source Code 13 Key of a requirement with program logic in `requirements.json` 37
Source Code 14 Filtered Requirements which are not from gems' source code . 38
Source Code 15 Commit data signed by git 40
Source Code 16 Generate SSH key pair & append to git's `allowed_signers` . 45
Source Code 17 `.git/config` file enabling Dilithium commit signatures . . . 46
Source Code 18 Verification of Dilithium commit signature in git 46
Source Code 19 Show all changes to `ssh_data` 48
Source Code 20 Handling of Dilithium5 keys with `ssh_data` 49
Source Code 21 Handling of Dilithium5 certificates with `ssh_data` 50
Source Code 22 Handling of Dilithium5 signatures with `ssh_data` 50
Source Code 23 Commit data to be signed 51
Source Code 24 Verify Dilithium5 git commit signature with `ssh_data` . . . 51

Acronyms

Acronym	Description	Pages
AES	Advanced Encryption Standard	2
AIVD	Netherlands National Communications Security Agency	22, 24, 27
ANSSI	French National Agency for the Security of Information Systems	24
API	Application Programming Interface	25–27, 29, 52, 53
AST	Abstract Syntax Tree	30, 33, 34, 72
AVX	Advanced Vector Extensions	41
BSI	German Federal Office for Information Security	3, 17, 24, 25
CA	Certification Authority	22
CBOM	Cryptography Bill of Materials	26, 52
CG	Call Graph	30
CI/CD	Continous Integration / Continous Delivery	18, 26, 52
CIC	Cryptographic Inventory Compilation	21, 25–29, 52, 70
CPU	Central Processing Unit	41
CVE	Common Vulnerabilities and Exposures	25
CVP	Closest Vector Problem	12
CWI	Netherlands National Research Institute for Mathematics and Computer Science	22, 24, 27
DAST	Dynamic Application Security Testing	53
DH	Diffie-Hellman	2, 19, 23
DLP	Discrete Logarithm Problem	2, 3, 6, 71
DNS	Domain Name System	2
DSA	Digital Signature Algorithm	2, 19, 47
EC	Elliptic Curve	23
ECC	Elliptic Curve Cryptography	2
ECDH	Elliptic Curve Diffie-Hellman	6, 19
ECDSA	Elliptic Curve Digital Signature Algorithm	19, 40, 47
ETSI	European Telecommunications Standards Institute	21, 22, 24, 26
FIPS	Federal Information Processing Standard	23, 53
GDP	General Decoding Problem	9, 10
GPG	GNU Privacy Guard	39, 40
IKEv2	Internet Key Exchange Protocol Version 2	16
IPSec	Internet Protocol Security	16
KEM	Key Encapsulation Mechanism	1, 6, 11, 16
KEX	Key Exchange Mechanism	6, 16, 19, 45
LWE	Learning with Errors	1, 13, 14
MAC	Message Authentication Code	6
MACsec	Media Access Control Security	16
NCCoE	US National Cybersecurity Center of Excellence	17, 23, 25, 26

Acronym	Description	Pages
NIST	US National Institute for Standards and Technology	16, 17, 23–28, 41, 47, 53
PKI	Public Key Infrastructure	2
PQC	Post Quantum Cryptography	4–6, 16, 17, 20–24, 26–28, 39–41, 45, 47, 52, 53, 70
QC	Quantum Computer	2–6, 16, 17, 20, 21, 23, 71
QKD	Quantum Key Distribution	4
RFC	Request for Comments	19, 39, 45, 48
RSA	Rivest-Shamir-Adleman	2, 4, 19, 39, 40, 47, 71
S/MIME	Secure / Multipurpose Internet Mail Extensions	2
SAST	Static Application Security Testing	25–29, 33, 52
SBOM	Software Bill of Materials	25, 26
SCA	Software Composition Analysis	25–29, 52
SHA	Secure Hash Algorithm	2, 22
SIVP	Shortest Independent Vectors Problem	12, 13
SPA	Static Program Analysis	25, 37
SSH	Secure Shell	2, 16, 19, 39, 40, 45, 46, 48–50, 52, 72
SVP	Shortest Vector Problem	12, 13
TLS	Transport Layer Security	2, 16, 19
TNO	Netherlands Organisation for Applied Scientific Research	22, 24, 27
UTF-16	16 Bit Unicode Transformation Format	37
UTF-8	8 Bit Unicode Transformation Format	37, 52

Notation

Symbol	Definition	Pages
$\mathbb{Z}$	The set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	7
$\mathbb{Z}_m$	The factor ring $\mathbb{Z}/m\mathbb{Z}$	7
$\mathbb{F}_p$	The Galois Field with p elements for p prime	7
$\mathrm{im}_{\mathbb{F}}$	The image of a linear map over a field $\mathbb{F}$	7
ker	The kernel of a map	7
$\mathbb{N}$	The set of natural numbers without 0	8
$\mathbb{N}_0$	$\mathbb{N} \cup \{0\}$	8
$[n]$	The set $\{x \in \mathbb{N}_0 \mid x \leq n\}$	8
$\mathbb{F}[x]$	The algebra of polynomials over a field $\mathbb{F}$ with one variable x	10
$a \equiv b \mod c$	For elements $a, b, c \in R$ of a ring R, $a \equiv b \mod c :\Leftrightarrow a - b \in (c)$	10
$gcd(x, y)$	Greatest common divisor of x and y	10
$\langle S \rangle_C$	Linear span of a subset S of a $\mathbb{F}$-vector space with coefficients in $C \subset \mathbb{F}$	11