UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Cybersecurity Curriculum

Semjon Kravtšenko

Efficient Two-Party ML-DSA Protocol in Active Security Model

Master's Thesis (21 ECTS)

Supervisors: Peeter Laud, PhD
Toomas Krips, PhD

Tartu 2025

Efficient Two-Party ML-DSA Protocol in Active Security Model

Abstract:
ML-DSA is a NIST standard that defines a signature scheme: a set of algorithms for creating and verifying digital signatures. Digital signatures can be used, for example, to authenticate to websites online and to sign documents. ML-DSA signatures, unlike signatures that follow so-called classical formats, are quantum-resistant: it is believed that forging ML-DSA signatures is inviable even with a cryptographically relevant quantum computer (that is not yet known to exist).
The security of a signing scheme relies on the secrecy of the used private key material. One way to increase the security of a signing scheme is to distribute the secret material across multiple devices, such that a sufficient number of them need to cooperate to create a signature. One scheme, that distributes the key across two devices, is implemented in SplitKey® technology, which is used in a popular signing solution Smart-ID® [1]. Unfortunately, a two-party scheme that could create standards-compliant quantum-resistant signatures does not exist.
This thesis presents a novel two-party signing scheme capable of creating ML-DSA-compliant signatures — Duolithium. This scheme is resistant against potential active attacks by either party, both during the key generation and signing processes. The thesis proposes some parts of Duolithium that were invented as a part of this thesis research and documents the remaining parts with reliance on prior research. Additionally, this thesis presents a complete, tested for functionality implementation of Duolithium in Python, together with the results of the benchmarks of network overhead and computational performance. The benchmark results suggest that Duolithium may be used to implement a new, quantum-resistant version of SplitKey that would be fully compatible with any signature verification component that supports ML-DSA.

Keywords:
ML-DSA, MPC, active security, cryptography, post-quantum cryptography

CERCS:
P170 Computer science, numerical analysis, systems, control

Efektiivne kaheosapoolne ML-DSA protokoll aktiivse turvalisusega

Lühikokkuvõte:
ML-DSA on NISTi standard, mis defineerib teatud signatuuriskeemi, s.t. algoritmid digitaalsignatuuride loomiseks ja kontrollimiseks. Digitaalsignatuure saab kasutada näiteks veebilehtedel autentimiseks ja dokumentide allkirjastamiseks. ML-DSA signatuurid, erinevalt signatuuridest, mis järgivad nn klassikalisi formaate, on kvantarvutikindlad: usutakse, et ML-DSA signatuuri võltsimine on praktikas võimatu isegi krüptograafiliselt olulise kvantarvuti abil (mida teadolevalt veel ei eksisteeri).
Signatuuriskeemi turvalisus sõltub kasutatud privaatvõtme materjali salajasusest. Üks viis signatuuriskeemi turvalisemaks muuta on privaatvõtme jagamine mitme seadme vahel, nii et allkirja loomiseks peavad piisavalt paljud neist koostööd tegema. Üks skeem, mis jagab võtme kahe seadme vahel, on rakendatud SplitKey® tehnoloogias, mis on kasutusel populaarses autentimis- ja allkirjastamislahenduses Smart-ID® [1]. Kahjuks ei eksisteeri kahe osapoolega skeemi, mis suudaks luua mõnele standardile vastavaid kvantarvutikindlaid allkirju.
Käesolev töö kirjeldab Duolithiumi — kahe osapoolega signeerimisskeemi, mis suudab luua ML-DSA standardile vastavaid signatuure. See skeem on vastupidav võimalikele aktiivsetele rünnetele kummaltki osapoolelt, nii võtme genereerimise kui ka signeerimise protsesside ajal. Meie töö pakub välja Duolithiumi teatud alamprotokollide konstruktsioonid ning dokumenteerib ülejäänud osa skeemist tuginedes eelnevatele teadusuuringutele. Samuti esitab meie töö Duolithiumi täieliku realisatsiooni Pythonis, ühes funktsionaalsuse testide ning võrguliikluse ja arvutusliku jõudluse mõõtmistega. Mõõtmistulemused viitavad sellele, et Duolithium võib olla sobiv sellise kvantarvutikindla SplitKey versiooni loomiseks, mis oleks täielikult ühilduv kõigi ML-DSA-d toetavate signatuuriverifitseerimiskomponentidega.

Võtmesõnad:
ML-DSA, mitme osapoolega arvutus, aktiivne turvalisus, krüptograafia, postkvantkrüptograafia

CERCS:
P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction

Estonia is recognized for its innovative electronic solutions that enable its residents to perform financial transactions, interact with governmental services and even participate in elections, all via the internet. One such solution is built on SplitKey [2]. SplitKey is a threshold cryptography technology that allows creating digital signatures using a private key that is sharded (split) across two devices. This prevents either party from using the private key without the cooperation of the other party. A potential attacker would need to compromise both devices to extract the private key.

The original SplitKey creates RSA-compatible signatures. Since the RSA algorithm is *not quantum-resistant*, RSA-compatible signatures do not provide the desired level of protection against forgeries in the presence of an attacker with a cryptography-relevant quantum computer. Such a computer is not yet known to exist, but advancements in quantum computing could make it feasible in the future. In a 2022 survey of 40 leading experts in the field, more than half found that the likelihood of a quantum computer being able to break RSA-2048 within fifteen years is around 50% or more [3, Figure 6]. Additionally, in November 2024, NIST published a draft [4] for a recommendation to discontinue the use of non-quantum-resistant digital signatures in the USA governmental institutions by 2035. The survey and the recommendation underscore the urgent need for transitioning to post-quantum cryptographic standards, due to both security considerations and regulatory requirements.

Development of threshold schemes that would produce signatures compatible with standardized and widely-supported cryptographic algorithms is an ongoing pursuit. This thesis describes a novel Multi Party Computation based threshold scheme for two parties (named Duolithium) that creates signatures compatible with ML-DSA, which *are quantum-resistant*. This scheme is secure against key extraction attacks and signature forgeries, even in the case of active attacks by the Server or the Client. To function, Duolithium relies on a third participant — the Correlated Randomness Provider (abbreviated as CRP).

1.1 ML-DSA

ML-DSA is a digital signature algorithm, standardized by NIST in FIPS 204 [5] in August 2024. ML-DSA is a lattice-based quantum-resistant algorithm: it is believed that even with a sufficiently powerful quantum computer, it would be infeasible to forge an ML-DSA signature. The first version of the algorithm later standardized as ML-DSA was proposed [6] in 2017 under the name CRYSTALS-Dilithium.

ML-DSA variants (parameter sets) are ML-DSA-44, ML-DSA-65, and ML-DSA-87. These variants provide different claimed security strength levels (2, 3 and 5, respectively) and their construction is different only in the values of some constants. The "weakest" of these levels (level 2) is defined [7] to be met if any attack that breaks the relevant security definition (for ML-DSA: signature unforgeability) must require computational resources comparable to or greater than those required for collision search on a 256-bit hash function. The "strongest" level is tied [7] to the difficulty of a key search on a block cipher with a 256-bit key.

1.2 MPC

Multiparty computation (MPC) is a branch of cryptography that deals with the design and analysis of algorithms that allow multiple parties to compute a function over their inputs while keeping those inputs private. A special case of MPC is two-party computation (2PC). In the context of this thesis, two-party computation allows creating ML-DSA signatures without either party possessing the corresponding private key.

1.3 Scope and contributions

This thesis documents the protocols that are required for implementing a 2PC ML-DSA scheme, while also providing some functionality and security proofs. The described protocols can be divided into three categories:

A. General-purpose 2PC protocols that are already published

B. ML-DSA-specific protocols that were proposed [8] (but not published) before the commencement of the research work for this thesis

C. ML-DSA-specific protocols that were invented as a part of this thesis research

Thus, the main research question of this thesis is: "Is it possible to create a practical 2PC system, capable of producing ML-DSA-compatible signatures?" The engineering research method was employed to develop prototype programs that implement the ML-DSA key generation and signature creation protocols (according to how they are presented in this thesis). Experimentally, these programs were tested for functionality and benchmarked for computational performance and for used traffic. The empiric benchmark results were used to answer the main research question.

1.4 Structure of the thesis

The thesis is structured as follows. Chapter 2 outlines the aspects of ML-DSA and MPC that are relevant to the described scheme, as well as the necessary general-purpose 2PC protocols. Chapter 3 documents the unpublished subprotocols for 2PC ML-DSA. Chapter 4 describes the subprotocols for 2PC ML-DSA that were created as part of the research for this thesis. Chapter 5 documents the unpublished ML-DSA key generation and signing protocols, which leverage the protocols discussed in the previous three chapters. Chapter 6 provides a general description of the prototype implementation, as well as performance and traffic measurements for this implementation. Chapter 7 points out the potential application of the scheme described in this thesis and outlines the topics for further research.

2 Background

This chapter provides the necessary background for the following chapters of the thesis. First, the used mathematical objects and operations on them are explained. Then, ML-DSA key generation and signing algorithms are described, with focus on details relevant for creating the corresponding 2PC protocols. Then, the basics of 2PC are outlined, along with some supporting subprotocols that are used in Duolithium.

2.1 Polynomial ring

This section describes the relevant mathematical background: modulo rings $\mathbb{Z}_q$ and operations on them; polynomial rings $\mathbb{Z}_q[X]/(X^n + 1)$, polynomial addition, multiplication and reduction; NTT-form of a polynomial.

Modular arithmetic. The operation of taking the *modulo* of an integer a under modulus q ($q \neq 0$) is denoted as $a \bmod^+ q$ and is defined to return the unique integer r, for which $0 \leq r < q$ and there exists an integer[1] n, such that $a = q \cdot n + r$. Additionally, lets define $a \bmod^\pm q = r$, where r is the unique integer $-\lceil \frac{q}{2} \rceil < r \leq \lfloor \frac{q}{2} \rfloor$, such that $r \bmod^+ q = a \bmod^+ q$ [5, Page 6]. Addition of integers under modulus q is equivalent to adding these integers and then taking the modulo of the result under modulus q. Multiplication of integers under modulus q is equivalent to multiplying these integers and then taking the modulo of the result under modulus q.

Modulo ring. A *ring* is a set with two binary operations on the elements of the set — addition (denoted as $a + b$) and multiplication (denoted as $a \cdot b$) — such that for all a, b, c in this ring:

1. $a + b = b + a$ (addition is commutative)
2. $(a + b) + c = a + (b + c)$ (addition is associative)
3. There exists an additive identity 0. That is, for every a: $a + 0 = a$
4. For every a there exists an additive inverse $-a$, such that $a + (-a) = 0$
5. $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ (multiplication is associative)
6. $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(b + c) \cdot a = b \cdot a + c \cdot a$ (multiplication is distributive over addition)

[1] This integer n may be positive, zero, or negative.

Note that it is implied that both addition and multiplication produce the elements of the same set. Subtraction of b from a is defined as $a - b = a + (-b)$. One example of a ring is a set of integers modulo q, with operations of addition and multiplication under modulus q. Such a ring is called a *modulo ring* and denoted as $\mathbb{Z}_q$. In this ring, there exists the unique multiplicative identity element, denoted as 1, such that $1 \cdot a = a$ for every a; and multiplication is commutative: for all a, b in this ring $a \cdot b = b \cdot a$. [9, Chapter 12]

A *field* is a ring with commutative multiplication and multiplicative identity 1, where every element $a \neq 0$ has a multiplicative inverse a^{-1}. Every modulo ring $\mathbb{Z}_q$, where q is a prime, is a field. [9, Chapter 13]

Polynomial ring. The ring of *polynomials* over R, denoted as $\mathbb{Z}_q[X]$ is the following mathematical object:

$$\mathbb{Z}_q[X] = \{a_n \cdot X^n + a_{n-1} \cdot X^{n-1} + \cdots + a_1 \cdot X + a_0 \mid a_i \in \mathbb{Z}_q\},$$

where necessarily $n \geq 0$. Here X is a formal variable, thus the write-up $a_n \cdot X^n + \cdots + a_1 \cdot X + a_0$ really denotes the tuple $(a_n, \ldots, a_1, a_0)$, while reflecting how the operations in a ring of polynomials are defined. Suppose $f = a_n \cdot X^n + \cdots + a_0$ and $g = b_n \cdot X^n + \ldots b_0$ belong to $\mathbb{Z}_q[X]$. Then their sum is

$$f + g = (a_n + b_n) \cdot X^n + \cdots + (a_0 + b_0)$$

and their product is

$$f \cdot g = \sum_{i=0}^{n-1} \sum_{u=0}^{n-1} (a_i \cdot b_u) \cdot X^{u+i}$$

(results of both operations also belong to $\mathbb{Z}_q[X]$). The operation of taking the modulo of a polynomial f under modulus $g = b_n \cdot X^n + \cdots + b_0$ ($g \neq 0$) is denoted as $f \bmod g = r$ and is defined to return the unique integer polynomial $r = c_m \cdot X^m + \cdots + c_0$, such that $m < n$ and there exists $t \in \mathbb{Z}_q[X]$, for which $f = g \cdot t + r$. [9, Chapter 16]

Notation $\|p\|_\infty$ denotes the infinity norm of a polynomial. For $p \in \mathbb{Z}_q[X]$, $p = a_n \cdot X^n + \cdots + a_0$: $\|p\|_\infty = \max_{i=0}^{n-1} |a_i \bmod^{\pm} q|$. Infinity norm of a vector of polynomials is the maximum of the infinity norms of these polynomials.

Let $\mathbb{Z}_q[X]/(X^n + 1)$ be defined as the set of polynomials modulo $g = X^n + 1$ with two operations: addition modulo g and multiplication modulo g. As can be verified, this mathematical object is a ring. In ML-DSA, polynomials that are used are in such a ring, with fixed q and n values, which are $q = 8380417$ and $n = 256$; henceforth in this work, only such polynomials are considered.

Notice that the multiplication of two polynomials in the ring $\mathbb{Z}_q[X]/(X^n + 1)$ may be performed by first multiplying them (per the formula above, considering them to be elements of $\mathbb{Z}_q[X]$) and then reducing the product. This is inefficient; in ring $\mathbb{Z}_q[X]/(X^n + 1)$, the computational complexity of such multiplication is $O(n^2)$.

Number Theoretic Transform. There is a more efficient way to multiply elements of $\mathbb{Z}_q[X]/(X^n+1)$, with computational complexity of $O(n \log n)$, by using *Number Theoretic Transform* (NTT) — a type of Discrete Fourier Transform. To multiply polynomials f and g in $\mathbb{Z}_q[X]/(X^n+1)$ using NTT, first, they must be converted to the so-called NTT-form (this step has the complexity of $O(n \log n)$). Then, the NTT forms of f, g are multiplied coefficient-wise (has $O(n)$ complexity). The result is the NTT-form of the $f \cdot g$. This result can then be converted to the "normal" form (always possible, since NTT is a bijection; has $O(n \log n)$ complexity). NTT is linear: for any a and b in R,

$$a \cdot \text{NTT}(f) + b \cdot \text{NTT}(g) = \text{NTT}(a \cdot f + b \cdot g).$$

Throughout this work, the NTT operations are performed implicitly whenever required. The NTT forms have the same lengths as "normal" forms and are likewise stored in computer memory as arrays of 256 integer values. More information on NTT can be obtained from *Satriawan et. al.* [10].

2.2 ML-DSA key generation

ML-DSA is a signature scheme, meaning that the purpose of the key generation algorithm is to produce a keypair, where one key is the private key, that allows creating signatures for specific messages (or, digests), and the other key is the public key, that allows verifying that a given signature was indeed produced using the corresponding private key.

The simplified version of the ML-DSA key generation algorithm is shown in Figure 1. For the concrete values of the constants q, l, k and η, which are seen in the algorithm, refer to the upper part of Table 1.

Gen

01: $A \xleftarrow{\$} [\mathbb{Z}_q[X]/(X^n+1)]^{k \times \ell}$
02: $(\boldsymbol{s}_1, \boldsymbol{s}_2) \xleftarrow{\$} S_\eta^\ell \times S_\eta^k$
03: $\boldsymbol{t} := A\boldsymbol{s}_1 + \boldsymbol{s}_2$
04: **return** $(pk = (A, \boldsymbol{t}), sk = (A, \boldsymbol{t}, \boldsymbol{s}_1, \boldsymbol{s}_2))$

Figure 1. Key generation algorithm (adapted) [6, Figure 1].

The domain S_η is defined as $S_\eta := \{p \in \mathbb{Z}_q[X]/(X^n+1) \mid \|p\|_\infty \leq \eta\}$.

ML-DSA employs several optimizations to reduce the size of the public key. Firstly, the matrix A is actually generated from a seed $\rho \in \mathbb{Z}_2^{256}$ and only this seed is included in the public key. This allows for saving $24 \cdot l \cdot k \cdot n - 256$ bits. Secondly, not the entire value of $\boldsymbol{t}$ is saved, but only the "high bits" $\boldsymbol{t}_1$, where $\boldsymbol{t}_1 = (\boldsymbol{t} - \boldsymbol{t} \bmod^{\pm} 2^d)/(2^d)$, with $d = 13$. This allows for saving $d \cdot k \cdot n$ bits. However, due to this optimization, it is necessary

Table 1. Parameters defined by the ML-DSA parameter sets.

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87
q - integer modulus	8380417	8380417	8380417
(l, k) - dimensions of A	$(4, 4)$	$(6, 5)$	$(8, 7)$
η - for private key generation	2	4	2
γ_1 - for obtaining $\boldsymbol{y}$	2^{17}	2^{19}	2^{19}
γ_2 - for Decompose	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
τ - for procedure H	39	49	60
$\beta = \tau \cdot \eta$	78	196	120

for the signing algorithm to "aid" the verification algorithm by including additional information in the signature — so-called hints (discussed in the next Section). These hints do not depend on the secret material.

Additionally, the secret values $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ in the ML-DSA key generation algorithm are derived deterministically from a seed ρ', rather than being generated directly. The seeds ρ and ρ' are derived from the random seed ζ. The private key contains an additional value K (also generated from ζ), that influences the formation of the masking vector during signing (see the next Section).

2.3 ML-DSA signing

The simplified version of the ML-DSA signature creation algorithm is described in Figure 2. The values of various constants are provided in Table 1.

Sign(sk, M**)**
01: $\boldsymbol{z} := \perp$
02: **while** $\boldsymbol{z} = \perp$ **do**
03: $\quad \boldsymbol{y} \xleftarrow{\$} S^{\ell}_{\gamma_1 - 1}$
04: $\quad \boldsymbol{w} := A\boldsymbol{y}$
05: $\quad \boldsymbol{w}^H := \text{HighBits}(\boldsymbol{w}, 2\gamma_2)$
06: $\quad c \in B_\tau := H(M | \boldsymbol{w}^H)$
07: $\quad \boldsymbol{z} := \boldsymbol{y} + c\boldsymbol{s}_1$
08: $\quad \boldsymbol{r} := \boldsymbol{w} - c\boldsymbol{s}_2$
09: $\quad \boldsymbol{r}^L := \text{LowBits}(\boldsymbol{r}, 2\gamma_2)$
10: $\quad$ **if** $\|\boldsymbol{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\boldsymbol{r}^L\|_\infty \geq \gamma_2 - \beta$
11: $\quad\quad \boldsymbol{z} := \perp$
12: **return** $\sigma = (\boldsymbol{z}, c)$

Figure 2. Signing algorithm (adapted) [6, Figure 1].

The signing algorithm uses HighBits and LowBits algorithms, both of which are defined in terms of Decompose (please see Figure 3). For inputs (v, α), Decompose returns the unique pair of values (v^H, v^L), such that $v = \alpha \cdot v^H + v^L$, whereas $0 \le v^H < \lfloor \frac{q}{\alpha} \rfloor$ and $-\lceil \frac{\alpha}{2} \rceil < v^L \le \lfloor \frac{\alpha}{2} \rfloor$, or $(v^H, v^L) = (0, -\lceil \frac{\alpha}{2} \rceil)$. The design decision to include the second option was necessitated by the need to ensure that Decompose is defined for $v = q - \lfloor \frac{\alpha}{2} \rfloor$; Decompose can only be used with values α, such that $q \bmod^+ \alpha = 1$. For inputs (v, α), both HighBits and LowBits perform Decompose and return either v^H, or v^L, correspondingly. In the Figure 2, HighBits and LowBits are applied coefficient-wise. Please note that $\boldsymbol{t}_1$ (see the previous Section) is obtained using a different procedure — Power2Round (omitted for brevity).

Decompose(v, α)
01: $q = 8380417$
02: $r := r \bmod^+ q$
03: $r^L := r \bmod^\pm \alpha$
04: **if** $r - r^L = q - 1$
05: $\quad r^H := 0$
06: $\quad r^L := r^L - 1$
07: **else**
08: $\quad r^H = \frac{r - r^L}{\alpha}$
09: **return** (r^H, r^L)

Figure 3. Decompose.

The procedure H in the Figure 2 obtains (using a hash function) a 256-bit digest from the binary representation of $(M|\boldsymbol{w}^H)$ and uses it to sample c from the domain of polynomials $\{f = a_{n-1} \cdot X^{n-1} + \cdots + a_0 \mid f \in \mathbb{Z}_q[X]/(X^n + 1), \sum_{i=0}^{n-1} |a_i| = \tau, \|f\|_\infty = 1\}$. The standardized (non-simplified) algorithm derives the digest from $(\mu|\boldsymbol{w}^H)$, where μ depends on the hash of the public key tr and the message M.

The masking vector $\boldsymbol{y}$ is considered secret material, since obtaining it along with the corresponding signature and the public key allows reconstructing the private key with negligible computational effort. In the actual ML-DSA signing algorithm, $\boldsymbol{y}$ is derived deterministically from the value K (stored in the private key), the optional randomness value rnd (supplied to the signing algorithm) and the value μ. [2]

The operation on line 10 is called rejection sampling: the value z may be revealed only if the corresponding rejection sampling does not fail[3]. Due to the rejection sampling, multiple signing attempts are typically required. According to Table 1 of FIPS 204 [5], the *while* loop in the algorithm runs, on average, 4.25 times for ML-DSA-44, 5.1 times for ML-DSA-65 and 3.85 times for ML-DSA-87.

As mentioned in the previous section, the signing algorithm incorporates extra information — hints — in the signature. For performance, the original ML-DSA signing algorithm uses $\boldsymbol{s}_2$ to generate hints, although hints do not depend on $\boldsymbol{s}_2$ (or other secret values). The number of hints that need to be encoded varies. This number does not impact the security properties of the signature. However, since FIPS 204 defines a constant size for the hints object, hint creation fails if too many are required. The probability of failure was heuristically estimated [11, Section 3.4] to be below 2%. [4]

[2]Thus, ML-DSA signing can operate in the deterministic mode, without relying on a quality randomness source, as long as K has high entropy and is kept secret.

[3]Revealing $\boldsymbol{z}$ after successful rejection sampling is allowed, even if the make hint procedure would fail.

[4]The estimation was originally obtained for CRYSTALS-Dilithium Version 3.1, but it also holds for ML-DSA, since the algorithms were not changed in a way that would change this probability.

2.4 ML-DSA signature verification

For completeness, we provide the simplified version of the ML-DSA signature verification algorithm, in Figure 4. The HighBits operation is applied coefficient-wise.

Verify(pk, M**,** $\sigma = (\boldsymbol{z}, c)$**)**
01: $\boldsymbol{w}^{H\prime} := \text{HighBits}(A\boldsymbol{z} - c\boldsymbol{t}, 2\gamma_2)$
02: **return** $\|\boldsymbol{z}\|_\infty < \gamma_1 - \beta$ **and** $c = H(M \| \boldsymbol{w}^{H\prime})$

Figure 4. Signature verification algorithm (adapted) [6, Figure 1].

The correctness of the algorithm relies on the fact that $A\boldsymbol{z} = A\boldsymbol{y} - c\boldsymbol{s}_2 = \boldsymbol{r}$. Since $\|\boldsymbol{r}^L\|_\infty < \gamma_2 - \beta$: $(\boldsymbol{r} + c \cdot \boldsymbol{s}_2)^H = \boldsymbol{r}^H$, because $\|c \cdot \boldsymbol{s}_2\|_\infty \leq \beta$, which means that adding $c \cdot \boldsymbol{s}_2$ to $\boldsymbol{r}$ does not change the value, such that it would affect $\boldsymbol{r}^H$. Thus, $\boldsymbol{w}^{H\prime} = \boldsymbol{w}^H$, where $\boldsymbol{w}^H$ was obtained using the signing algorithm. Please refer to FIPS 204 for a complete proof of correctness.

The actual algorithm derives A from the seed ρ (included in the public key). Rather that applying HighBits to compute $\boldsymbol{w}^{H\prime}$, it instead obtains $\boldsymbol{w}^{H\prime}$ by using $\boldsymbol{t}_1$ (from the public key) and the hints object h (included in the signature).

The purpose of Duolithium is to create signatures that would be successfully verified by the original (complete, non-simplified) ML-DSA signature verification algorithm. Therefore, it is crucial for the design of Duolithium that both the signature and the public key objects are formed precisely in the formats expected by the verification algorithm.

2.5 Additive sharing

One fundamental technique of MPC is secret sharing, which involves distributing a secret value among multiple participants, such that no participant can recover the shared value without cooperation from another party (or parties). A straightforward example of secret sharing is *additive sharing*. An additive sharing of value v in a finite ring R to n shares is a tuple $(\llbracket v \rrbracket_0, \llbracket v \rrbracket_1, \ldots, \llbracket v \rrbracket_{n-1})$, such that $v_i \in R$ for all $i \in [0; n)$ and $v = \sum_{i=0}^{n-1} \llbracket v \rrbracket_i$. Each individual $\llbracket v \rrbracket_i$ is called *a share*. To enter a value v as an input in an additive sharing scheme, the party i that possesses that value generates and sends to the other parties shares $\llbracket v \rrbracket_u \xleftarrow{\$} R$ $(u \neq i)$, and obtains its share as $\llbracket v \rrbracket_i \leftarrow v - \sum_{j \in \{1,\ldots,n\} \setminus i} \llbracket v \rrbracket_j$. Thus, even if $n - 1$ parties collude, they cannot obtain any additional information about the value v from their shares.

In a 2PC scheme, the values of which the privacy should be protected are shared to two shares, where each party possesses one share. Duolithium is a 2PC scheme; the parties are named Server (has index 0) and Client (has index 1).

In Duolithium, the values are additively shared in integer rings. Specifically, suppose that a value v that is additively shared in an integer ring $\mathbb{Z}_Q$. Then, $\llbracket v \rrbracket_0$ and $\llbracket v \rrbracket_1$ are also

elements of ring $\mathbb{Z}_Q$. An element of ring $\mathbb{Z}_Q$ can take on an integer value between 0 and $Q-1$ (inclusive).

The following operations on shares can be performed locally[5]:

1. Constant introduction. A sharing of a value v known by both parties can be obtained as follows: $[\![v]\!] = (v, 0)$. Constant addition is performed as shares addition, where one of the of the shares is an introduced constant.

2. Addition (denoted as $[\![a]\!] + [\![b]\!]$). Since $[\![a+b]\!] = [\![a+b]\!]_0 + [\![a+b]\!]_1 = [\![a]\!]_0 + [\![b]\!]_0 + [\![a]\!]_1 + [\![b]\!]_1 = [\![a]\!] + [\![b]\!]$, sharing $[\![a+b]\!]$ can be obtained as $([\![a]\!]_0 + [\![b]\!]_0, [\![a]\!]_1 + [\![b]\!]_1)$, where the necessary computations are local. Subtraction of two shares is performed as an addition of: the left share, the right share multiplied by -1 (see next).

3. Multiplication by a known constant (denoted as $c \cdot [\![v]\!]$ or $[\![v]\!] \cdot c$). Since $[\![c \cdot v]\!] = [\![c \cdot v]\!]_0 + [\![c \cdot v]\!]_1 = c \cdot [\![v]\!]_0 + c \cdot [\![v]\!]_1$, sharing $[\![c \cdot v]\!]$, where c is a value known by both parties, can be obtained as $(c \cdot [\![v]\!]_0, c \cdot [\![v]\!]_1)$, where the necessary computations are local.

A vector of shared values is denoted as $[\![\boldsymbol{v}]\!]$. The shared value at index u of this vector is denoted as $[\![v_u]\!]$. The corresponding vector of shares that belong to party i is denoted as $[\![\boldsymbol{v}]\!]_i$. Sum of two shared vectors of the same length (sum of two vectors of shared values) is defined as $[\![\boldsymbol{a}]\!] + [\![\boldsymbol{b}]\!] = ([\![a_0]\!] + [\![b_0]\!], [\![a_1]\!] + [\![b_1]\!], \dots)$. Multiplication of a shared vector by a constant is defined as $c \cdot [\![\boldsymbol{v}]\!] = (c \cdot [\![v_0]\!], c \cdot [\![v_1]\!], \dots)$.

Sharing of a polynomial is considered to be a shared vector of its coefficients. A shared polynomial f can be multiplied by a known polynomial g locally. First, the polynomials are converted to their NTT forms. Then, the NTT forms are multiplied to obtain the NTT form of $f \cdot g$. Finally, the result is converted to the "normal" form. Note that all performed operations — NTT, multiplication of a share by a known value and NTT^{-1} — are linear.

There exist protocols that allow performing non-linear computations on the shares, non-locally (e.g., see Section 2.8.2). The process by which the parties reconstruct a shared value from their shares involves sending shares (either from one party to another or from both parties to each other). This process is described in the next Section.

2.6 BeDOZa style MACs

Additively sharing a value does not provide protection against a potentially malicious party modifying their share. If any share of a shared value is tampered with, the output of a protocol that relies on it may be incorrect. Moreover, in some cases, the output may

[5]It can be shown that if the value of a function at a given point depends non-linearly on a shared value, then the value of the function at that point cannot be computed locally.

be unsafe to reveal, as it could expose information about the secret material, which is not allowed.

Therefore, to achieve security against a potentially malicious party, a certain mechanism is required that allows the other party to verify that the share it receives from the potentially malicious party is correct. A typical way of achieving this is through the use of "control values" that are revealed simultaneously with the share, the correctness of which they allow verifying. Such "control values" bear similarity to Message Authentication Codes (MACs); due to historical reasons, they are often referred to by the same label.

In Duolithium, BeDOZa style MACs [12] are used. For a value v, these MACs are the values of form $\Delta_0 \cdot [\![v]\!]_1$ (further referred to as Server MAC) and $\Delta_1 \cdot [\![v]\!]_0$ (further referred to as Client MAC). These values are additively shared between the parties. The Δ_i value is selected (generated) by party i from the same ring as the value v and is known only by that party and the CRP — an extra participant of the scheme, whose role will be explained in the next Section. Due to certain security considerations stemming from the security proofs for the properties of BeDOZa style MACs[6], the ring of a Δ value must necessarily be a field. Therefore, only values shared in fields can be MACed.

The Protocol 1 outlines the procedure the parties use to declassify (mutually reveal the value of) a vector of shared values. Note that a party accepts the shares of another party only if it also receives the hash value that is consistent with the correct MAC value for the received shares. The honest party aborts the protocol execution (and deletes the secret material) if the security check (on line 5) enabled by the use of MACs, fails[7]. The protocol can be modified in an obvious way to reveal the value to only one of the parties, or to declassify a value that does not have MACs.

Henceforth, we define the *actively secure sharing* of a value v as $\langle\!\langle v \rangle\!\rangle := ([\![v]\!], [\![\Delta_0 \cdot [\![v]\!]_1]\!], [\![\Delta_1 \cdot [\![v]\!]_0]\!])$, where $\Delta_i \cdot [\![v]\!]_{1-i}$ represents the MAC for a party i. Thus, $\langle\!\langle v \rangle\!\rangle$ denotes a share protected by the Server MAC and the Client MAC. For certain protocols (see Section 3.5), protection against potential active attacks is necessary only against the Client. Similarly, we define $[\![v \rangle\!\rangle := ([\![v]\!], [\![\Delta_0 \cdot [\![v]\!]_1]\!])$, so $[\![v \rangle\!\rangle$ denotes a share protected by the Server MAC alone (that is, the Server can detect if a potentially malicious Client tampered with the share).

In Protocol 1, $h(\dots)$ denotes a cryptographic hash function, a collision-resistant function, which produces a value in $\mathbb{Z}_2^{256}$ for a vector of values. Thus, using MACs does not significantly increase the traffic volume between the parties, as the traffic overhead from using MACs, for a vector of an arbitrary length, is constant.

Please note that the parties send the values asynchronously; i.e. neither party needs to wait to receive the shares of another party before sending own shares.

[6] Details are omitted for brevity.

[7] If Duolithium is incorporated into a PKI system, the certificate associated with the corresponding public key must be revoked upon the failure of any security check.

Protocol 1: `declassify`

Input: $\langle\!\langle \boldsymbol{v} \rangle\!\rangle$

Output: $\boldsymbol{v}$ (known by both parties)

1 $([\![\boldsymbol{v}]\!], [\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!], [\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]) \leftarrow \langle\!\langle \boldsymbol{v} \rangle\!\rangle$

	Server	**Client**
2	$h_0 \leftarrow h([\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_0)$	$h_1 \leftarrow h([\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_1)$
3	Send $[\![\boldsymbol{v}]\!]_0$ and h_0	Send $[\![\boldsymbol{v}]\!]_1$ and h_1
4	Receive $[\![\boldsymbol{v}]\!]'_1$ and h'_1	Receive $[\![\boldsymbol{v}]\!]'_0$ and h'_0
5	Verify $h(\Delta_0 \cdot [\![\boldsymbol{v}]\!]'_1 - [\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_0) \stackrel{?}{=} h'_1$	Verify $h(\Delta_1 \cdot [\![\boldsymbol{v}]\!]'_0 - [\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_1) \stackrel{?}{=} h'_0$

6 **Parties** accept that $[\![\boldsymbol{v}]\!]_0 + [\![\boldsymbol{v}]\!]'_1 = [\![\boldsymbol{v}]\!]'_0 + [\![\boldsymbol{v}]\!]_1 = \boldsymbol{v}$ and output $\boldsymbol{v}$

It is easy to see that if both parties are honest, the condition on line 5 is satisfied, since for every party i: $h'_i = h_i$ and $[\![\boldsymbol{v}]\!]'_i = [\![\boldsymbol{v}]\!]_i$, from which $h(\Delta_i \cdot [\![\boldsymbol{v}]\!]'_{1-i} - [\![\Delta_i \cdot [\![\boldsymbol{v}]\!]_{1-i}]\!]_i) = h(\Delta_i \cdot [\![\boldsymbol{v}]\!]_{1-i} - [\![\Delta_i \cdot [\![\boldsymbol{v}]\!]_{1-i}]\!]_i) = h([\![\Delta_i \cdot [\![\boldsymbol{v}]\!]_{1-i}]\!]_{1-i}) = h_{1-i} = h'_{1-i}$.

Let's show that if a malicious party i tampered with its share, with high probability it will be caught cheating during the declassification. Suppose that the party i sent $[\![\boldsymbol{v}]\!]'_i$ and $h'_i = h(m)$ to the honest party $1-i$ and that verify in line 5 passed. Therefore, since h is collision-resistant, with overwhelming probability $\Delta_{1-i} \cdot [\![\boldsymbol{v}]\!]'_i - [\![\Delta_{1-i} \cdot [\![\boldsymbol{v}]\!]_i]\!]_{1-i} = m$. If $[\![\boldsymbol{v}]\!]'_i \neq [\![\boldsymbol{v}]\!]_i$ (i.e., if i cheated), there exists t such that $[\![v_t]\!]'_i \neq [\![v_t]\!]_i$. Note that from the viewpoint of the malicious party, the following is a system of linear equations with two unknowns, Δ_{1-i} and $[\![\Delta_{1-i} \cdot [\![v_t]\!]_i]\!]_{1-i}$:

$$\begin{cases} \Delta_{1-i} \cdot [\![v_t]\!]'_i - [\![\Delta_{1-i} \cdot [\![v_t]\!]_i]\!]_{1-i} = m_t \\ \Delta_{1-i} \cdot [\![v_t]\!]_i - [\![\Delta_{1-i} \cdot [\![v_t]\!]_i]\!]_{1-i} = [\![\Delta_{1-i} \cdot [\![v_t]\!]_i]\!]_i \end{cases} .$$

By solving the system (which takes negligible computational effort), the malicious party can learn the value Δ_{1-i}. Thus, successfully performing cheating is not easier than obtaining the Δ_{1-i} value. In reality, this value Δ_{1-i} is chosen uniformly at random by the another party. The malicious party cannot guess it with probability better than $\frac{1}{Q}$, where Q is the size of the ring in which $\boldsymbol{v}$ is shared.

As follows from the argument above, MACs provide sufficient protection against tampering due to the unpredictability of the Δ values. If the field $\mathbb{Z}_q$ from which the Δ value is sampled is sufficiently large (if $q > 2^{128}$), the probability of successful cheating does not exceed 2^{-128}. However, if the field is not sufficiently large, $L = \lceil \frac{128}{\log_2(q)} \rceil$ independent Δ values should be used to ensure the desired bound on the probability of successful cheating ($p \leq 2^{-128}$). For example, for the ML-DSA $Q = 8380417$, 6 independently generated Δ_0 values are used for the Server MAC, and 6 independently generated Δ_1 values are used for the Client MAC. The Δ values used for different moduli are independent.

Please note that the protocols throughout this document are rendered, for brevity, to only note one value Δ per party.

Local operations from the previous section can also be performed on MACed shares.

1. Constant introduction: $\langle\!\langle v \rangle\!\rangle = (\llbracket v \rrbracket, \llbracket \Delta_0 \cdot \llbracket v \rrbracket_1 \rrbracket, \llbracket \Delta_1 \cdot \llbracket v \rrbracket_0 \rrbracket) = (v, v \cdot \Delta_0, 0) = ((v, 0), (v \cdot \Delta_0, 0), (0, 0))$, where v is a value known by both parties and Δ_0 is only known by the Server.
2. Addition: $\langle\!\langle a + b \rangle\!\rangle = (\llbracket a \rrbracket + \llbracket b \rrbracket, \llbracket \Delta_0 \cdot \llbracket a \rrbracket_1 \rrbracket + \llbracket \Delta_0 \cdot \llbracket b \rrbracket_1 \rrbracket, \llbracket \Delta_1 \cdot \llbracket a \rrbracket_0 \rrbracket + \llbracket \Delta_1 \cdot \llbracket b \rrbracket_0 \rrbracket)$.
3. Multiplication by a known constant: $\langle\!\langle c \cdot v \rangle\!\rangle = (c \cdot \llbracket v \rrbracket, c \cdot \llbracket \Delta_0 \cdot \llbracket v \rrbracket_1 \rrbracket, c \cdot \llbracket \Delta_1 \cdot \llbracket v \rrbracket_0 \rrbracket)$.

It is easy to show that the MACs on the results of the operations are correct. Operations on the shares without Client MACs are performed likewise.

2.7 Multiparty computation with CRP

The two-party computation protocols described in this thesis rely on a third participant — the Correlated Randomness Provider (CRP). The purpose of CRP is to emit (generate and distribute) the additively shared values to the parties (the Server and the Client). The values generated hold a specific relation (or relations) that are prescribed by the protocol, for which these values are used.

CRP never receives any information from the Server, or the Client — except for the randomly generated Δ values, that the CRP, alternatively, could generate itself. Also, honest CRP does not influence the outputs of the protocols. Thus the CRP is not considered to be a third party in the protocol.

As will become evident from the Chapter 6, the volume of data that the CRP needs to send to the parties is very large[8]. Thankfully, traffic to one of the parties can be significantly reduced by only providing this party with a seed, which can be expanded into a vector of shares. Let f be a cryptographically secure pseudorandom number generator, such that $f(s) = \langle\!\langle \boldsymbol{v} \rangle\!\rangle_1$. Then, $\boldsymbol{v} = \langle\!\langle \boldsymbol{v} \rangle\!\rangle_0 + f(s)$, where $\langle\!\langle \boldsymbol{v} \rangle\!\rangle_0$ will be sent to the Server and s is sent to the Client.

Throughout this thesis, the format and relation of the correlated randomness (CR) shares that are used in a protocol (if any) are specified in the description of that protocol, before the protocol body. After the CRP generates the necessary values according to the requirements, it represents them as vectors and shards every vector individually per Protocol 2.

[8] In the developed prototype, the CRP needs to send approximately 150MB of data (on average) to the Server, so the Server and the Client would be able to create one signature.

Protocol 2: Shares emission procedure (performed by the CRP)

Input: $\boldsymbol{v}$

1 $s \xleftarrow{\$} \mathbb{Z}_2^{256}$
2 $\langle\!\langle \boldsymbol{v} \rangle\!\rangle_1 \leftarrow f(s)$
3 $([\![\boldsymbol{v}]\!]_1, [\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_1, [\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_1) \leftarrow \langle\!\langle \boldsymbol{v} \rangle\!\rangle_1$
4 $[\![\boldsymbol{v}]\!]_0 \leftarrow \boldsymbol{v} - [\![\boldsymbol{v}]\!]_1$
5 $[\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_0 \leftarrow (\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1) - [\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_1$
6 $[\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_0 \leftarrow (\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0) - [\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_1$
7 $\langle\!\langle \boldsymbol{v} \rangle\!\rangle_0 \leftarrow ([\![\boldsymbol{v}]\!]_0, [\![\Delta_0 \cdot [\![\boldsymbol{v}]\!]_1]\!]_0, [\![\Delta_1 \cdot [\![\boldsymbol{v}]\!]_0]\!]_0)$
8 Send $\langle\!\langle \boldsymbol{v} \rangle\!\rangle_0$ to the Server and s to the Client

Shares that do not require MACs or only require Server MACs are emitted in a similar way.

Since the protocol flow in Duolithium is known in advance (except for the number of signing iterations, see Chapter 5), the CRP can generate the required values and "eagerly" stream them to the parties — before these values are needed — thus reducing the latency added by waiting for the CRP responses.

Additionally, instead of generating a new seed s for every shared vector, the CRP could derive all s values (using a cryptographically secure pseudorandom number generator) from a single seed. Moreover, this seed could be used for multiple protocol sessions (provided that the seeds s remain unpredictable to the Server).

2.8 General-purpose 2PC protocols

This subsection introduces several 2PC subprotocols that are used in Duolithium. All these are trivially extendable to handle shares with MACs.

2.8.1 Multiplication of shared values

Multiplication of shared values $u \in \mathbb{Z}_q$ and $v \in \mathbb{Z}_q$ can be computed using a Beaver triple [13], as described in Protocol 3. On line 4, according to the rules for adding a known value, only the Server adds the constant $-\alpha \cdot \beta$ to its share.

Protocol 3: `shares_multiplication`

Input: $[\![u]\!], [\![v]\!]$
Output: $[\![y]\!]$, such that $y = u \cdot v$
CR: $[\![a]\!], [\![b]\!], [\![c]\!]$, such that $a \xleftarrow{\$} \mathbb{Z}_q$, $b \xleftarrow{\$} \mathbb{Z}_q$ and $c \leftarrow a \cdot b$

1 $[\![\alpha]\!] \leftarrow [\![u]\!] - [\![a]\!]$
2 $[\![\beta]\!] \leftarrow [\![v]\!] - [\![b]\!]$
3 Parties declassify α and β
4 $[\![y]\!]_i \leftarrow \alpha \cdot [\![v]\!]_i + \beta \cdot [\![u]\!]_i + [\![c]\!]_i - \alpha \cdot \beta$

Proof of functionality. As can be verified, $y = u \cdot v$:

$$\begin{aligned} y &= [\![y]\!]_0 + [\![y]\!]_1 \\ &= \alpha \cdot v + \beta \cdot u + c - \alpha \cdot \beta \\ &= (u - a) \cdot v + (v - b) \cdot u + c - (u - a) \cdot (v - b) \\ &= u \cdot v - a \cdot v + u \cdot v - b \cdot u + c - u \cdot v + a \cdot v + b \cdot u - a \cdot b \\ &= u \cdot v + c - a \cdot b \\ &= u \cdot v \quad \square \end{aligned}$$

2.8.2 Characteristic vector

Characteristic vector (CV) of a value $v \in \mathbb{Z}_q$ is a vector $\boldsymbol{b}$ of length q, such that

$$b_i = \begin{cases} 1 & \text{if } i = v \\ 0 & \text{else} \end{cases}.$$

Henceforth, the notation $[\![v]\!]^{(q)}$ specifies that the value v is shared in the ring $\mathbb{Z}_q$, that is, $v \in \mathbb{Z}_q$; the superscript text merely provides additional information about the $[\![v]\!]$. Analogous notation is used for values with MACs and for vectors of shared values.

CV protocol [14], presented as Protocol 4, allows obtaining the shared characteristic vector $[\![\boldsymbol{b'}]\!]^{(Q)}$ of v from $[\![v]\!]^{(q)}$ and a desired modulus Q.

Protocol 4: `characteristic_vector`

Input: $[\![v]\!]^{(q)}$, Q
Output: $[\![\boldsymbol{b'}]\!]^{(Q)}$
CR: $[\![r]\!]^{(q)}$; $[\![\boldsymbol{b}]\!]^{(Q)}$, such that $r \xleftarrow{\$} \mathbb{Z}_q$ and $\boldsymbol{b}$ is the CV of r

1 $[\![d]\!]^{(q)} \leftarrow [\![v]\!]^{(q)} - [\![r]\!]^{(q)}$
2 Parties declassify d
3 $[\![\boldsymbol{b'}]\!]^{(Q)} \leftarrow \left([\![b_{0-d}]\!]^{(Q)}, [\![b_{1-d}]\!]^{(Q)}, \ldots, [\![b_{(q-1)-d}]\!]^{(Q)}\right)$ ▷ Indexes are *modulo* q

The volume of traffic between the CRP and Server, as well as the computational complexity for CRP, Server and Client grow linearly in the size of $\boldsymbol{b}$. Therefore, CV protocol is practical only for relatively small q.

Note that the characteristic vector $[\![\boldsymbol{b}]\!]^{(Q)}$ of a value $[\![v]\!]^{(q)}$ allows computing the value $f(v)$ of any function f (provided that f is computable for the domain of v): $[\![f(v)]\!]^{(Q)} = \sum_{i=0}^{q-1} f(i) \cdot [\![b_i]\!]^{(Q)}$. Values of multiple functions may be evaluated, but declassifying the value of one function may impact the secrecy of the other values.

2.8.3 Unmasked zero check for many values

Large zero check is a commonly known protocol that allows, from $[\![\boldsymbol{v}]\!]^{(q)}$, to obtain vector $[\![\boldsymbol{y}]\!]^{(q)}$ ($|\boldsymbol{v}|$ is much greater than $|\boldsymbol{y}|$), such that all elements of $\boldsymbol{y}$ equal zero if all elements of $\boldsymbol{v}$ equal zero, and (with overwhelming probability) there is a non-zero element in $\boldsymbol{y}$ if there is a non-zero element in $\boldsymbol{v}$. A version[9] of this protocol is presented as Protocol 5.

In the protocol, each party defines (via a seed) coefficients for c linear combinations for the values in $\boldsymbol{v}$ — these linear combinations are then computed by the parties. Thus, $|\boldsymbol{y}| = 2 \cdot c$. The parameter c is selected, such that $c \cdot \log_2 Q \geq 256$; $|\boldsymbol{y}| = 2 \cdot c$. This ensures that the probability of obtaining $[\![\boldsymbol{y}]\!]$, such that $\forall_{i=0}^{|\boldsymbol{y}|-1} y_i = 0$, but $\exists_{i=0}^{|\boldsymbol{v}|-1} v_i \neq 0$ is negligible.

Protocol 5: `large_zero_check_unmasked`

Input: $[\![\boldsymbol{v}]\!]$, c
Output: $[\![\boldsymbol{y}]\!]$

1 Party p generates seed $s_p \in \mathbb{Z}_2^{256}$
2 Parties exchange seeds, obtaining (s_0, s_1)
3 $M_0 \leftarrow f(s_0)$
4 $M_1 \leftarrow f(s_1)$
5 $[\![\boldsymbol{y}]\!] \leftarrow \begin{bmatrix} M_0 \\ M_1 \end{bmatrix} \cdot [\![\boldsymbol{v}]\!]$ ▷ Matrix multiplication

The function f used in Protocol 5 is a pseudo-random number generator, $f : \mathbb{Z}_2^{256} \rightarrow \mathbb{Z}_q^{c \times |\boldsymbol{v}|}$, where q is the modulus of the integer ring of v.

In practice, this protocol is used to reduce communication between the parties needed for verifying whether $[\![\boldsymbol{v}]\!]$ consists of zeros only. Since $[\![\boldsymbol{v}]\!]$ may be large, declassifying it directly would result in increased data transfer between the parties, compared to declassifying $[\![\boldsymbol{y}]\!]$.

Note that Protocol 5, as presented, is not *masked* — declassifying $[\![\boldsymbol{y}]\!]$ partially reveals information about $[\![\boldsymbol{v}]\!]$, in case $\exists_{i=0}^{|\boldsymbol{v}|-1} v_i \neq 0$.

Security-related zero check for many values

In Duolithium, unmasked zero checks for many values are only used for conducting "security checks", during which the parties prove that the sent shares equal zero and that they were not tampered with. Thus, an alternative protocol can be considered, where, for a shared vector $\boldsymbol{v}$, each party i computes the expected value of the other party's share $[\![\boldsymbol{v}]\!]'_{1-i} = 0 - [\![\boldsymbol{v}]\!]_i$; then parties send to each other and verify the appropriate MAC hashes. An important consideration of this protocol is that it would trigger the security failure if $\exists_{i=0}|\boldsymbol{v}| - 1 v_i \neq 0$, even if the MACs for the shares were correct.

[9] Many versions of this protocol exist, the version presented here matches the one used in the developed prototype (see Chapter 6).

3 Existing ML-DSA-specific protocols

This section documents and formalizes the Duolithium subprotocols that were described [8] prior the commencement of this research, but were not published.

Note that some protocols in this and the next Chapter are described (for clarity) as operating on a single value or returning one value. During the key generation or signing, parallel ("vectorized") versions of these protocols are employed, meaning multiple instances the same protocol are executed at the same time on different values.

3.1 Entry of shared values

Shares entry protocol (Protocol 6) is used by the parties to obtain $\langle\!\langle v \rangle\!\rangle$ from $[\![v]\!]$. Note that this protocol does not modify $[\![v]\!]$.

Protocol 6: `entry`

Input: $[\![v]\!]$
Output: $\langle\!\langle v \rangle\!\rangle$

	Server	**Client**
1	$([\![d_0]\!]_0, [\![w_0]\!]_0) \leftarrow (\Delta_0, [\![v]\!]_0)$	$([\![d_0]\!]_1, [\![w_0]\!]_1) \leftarrow (0, 0)$
2	$([\![d_1]\!]_0, [\![w_1]\!]_0) \leftarrow (0, 0)$	$([\![d_1]\!]_1, [\![w_1]\!]_1) \leftarrow (\Delta_1, [\![v]\!]_1)$
3	**for** $u \in \{0, 1\}$ **in parallel:**	
4	$[\![m_u]\!]$ = `shares_multiplication`$([\![d_u]\!], [\![w_{1-u}]\!])$	
5	$\langle\!\langle v \rangle\!\rangle \leftarrow ([\![v]\!], [\![m_0]\!], [\![m_1]\!])$	

This protocol is used during the key generation (in parallel for each coefficient of $[\![s_1]\!]$ and $[\![s_2]\!]$; see Section 4.1), to obtain MACs on the key shares that were generated by the parties.

3.2 Masking vector generation

The masking vector coefficient generation protocol (Protocol 7) allows the parties to generate $\langle\!\langle v \rangle\!\rangle$, such that $v \in (-2^L, \ldots, 2^L]$, for a specific value of L. This protocol is used during signing, in parallel for each coefficient of each polynomial of the masking vector.

Protocol 7: `generate_y`

Input: L

Output: $\langle\!\langle y \rangle\!\rangle^{(Q)}$

CR: $\langle\!\langle \boldsymbol{b} \rangle\!\rangle^{(Q)}$, such that $\boldsymbol{b} \xleftarrow{\$} \{0,1\}^{L+1}$

1 Party i generates $\boldsymbol{t}_i \xleftarrow{\$} \mathbb{Z}_2^{L+1}$

2 Parties send $\boldsymbol{t}_i$ to each other

3 $\boldsymbol{r} \leftarrow \boldsymbol{t}_0 + \boldsymbol{t}_1$

4 **for** $u \in \{0, \ldots, L\}$ **in parallel:**

5 $\quad \langle\!\langle p_u \rangle\!\rangle^{(Q)} \leftarrow \begin{cases} 1 - \langle\!\langle b_u \rangle\!\rangle^{(Q)} & \text{if } r_u = 1 \\ \langle\!\langle b_u \rangle\!\rangle^{(Q)} & \text{else} \end{cases}$

6 $\langle\!\langle y' \rangle\!\rangle^{(Q)} \leftarrow \sum_{u=0}^{L} 2^u \cdot \langle\!\langle p_u \rangle\!\rangle^{(Q)}$

7 $\langle\!\langle y \rangle\!\rangle^{(Q)} \leftarrow \langle\!\langle y' \rangle\!\rangle^{(Q)} - 2^L + 1$

Note that, assuming that the vector $\boldsymbol{b}$ (provided by the CRP) is correct, the distribution of v does not depend on the information provided by the CRP. Thus, the honest CRP does not learn any information about the masking vector $\boldsymbol{y}$ that may be used to create a created signature. This is a desirable property because information about $\boldsymbol{y}$ of a particular signature can be utilized in a key-extraction attack against that signature.

3.3 Asymmetric reshare to parts

This section documents a subprotocol of the rejection sampling protocol (see Section 3.5).

Let $\widetilde{x}$ for $x \in \mathbb{Z}_q$ be defined as an integer, whose value equals x; $0 \leq x < q$. Consider $(a, b) \in \mathbb{Z}_q$ for some q. Then, $a + b$ describes the addition of ring elements, which produces a ring element, the value of which is necessarily smaller than q. Conversely, $\widetilde{a} + \widetilde{b}$ describes the addition of two integers — the result of this addition may be equal to or greater than q.

Asymmetric reshare to parts protocol (Protocol 8) allows, from $[\![v]\!\rangle^{(Q)}$, radix r, output length d and output modulo q to obtain $[\![\boldsymbol{p}]\!\rangle^{(q)}$, with $|\boldsymbol{p}| = d$, such that $\widetilde{v} = \sum_{i=0}^{d-1} r^i \cdot \widetilde{p_i}$, or $\widetilde{v} + Q = \sum_{i=0}^{d-1} r^i \cdot \widetilde{p_i}$, while additionally $\forall_{i=0}^{d-1} p_i \leq 2 \cdot (r-1)$. The following input requirements must be satisfied: q is prime, $q > 2 \cdot (r-1)$, $r^d > 2 \cdot Q$.

In other words, the protocol "splices" the $[\![v]\!\rangle^{(Q)}$ into shared digits $[\![\boldsymbol{p}]\!\rangle^{(q)}$, least significant digit first. At each position, the digit can be "oversize": with a value of up to $2 \cdot (r-1)$. The value that was spliced can be either $\widetilde{v}$ or $\widetilde{v} + Q$ (since v can be represented by shares that add up to either of these values).

The local function `split`, for a value $v \in \mathbb{Z}$, and a vector of positive integers $\boldsymbol{c}$ (the "partition scheme"), obtains a vector of parts ("digits") $\boldsymbol{p}$, such that $\forall_{i=0}^{|\boldsymbol{c}|-1} 0 \leq p_i < c_i$ and $v = \sum_{i=0}^{|\boldsymbol{c}|-1} \left(p_i \cdot \prod_{u=0}^{i-1} c_u \right)$. In other words, the function converts an integer into a numeral system, which may have mixed base. It is easy to see that for a fixed $\boldsymbol{c}$, `split` is bijective. The local function `unsplit` is defined as the inverse function to `split`.

In Protocol 8, `split` is used once by the CRP and once by the Server, in both cases the partitioning scheme is $\boldsymbol{c} = [r] \times d$. In Duolithium, the values r, d, q for the asymmetric reshare to parts protocol are set as constants: $r = 15$, $d = 6$, $q = 29$.

Protocol 8: `reshare_to_parts_ap`

Input: $[\![v]\!\rangle^{(Q)}$

Output: $[\![\boldsymbol{p}]\!\rangle^{(q)}$

CR: m'_1 and $\boldsymbol{p}'_1$, such that $m'_1 \xleftarrow{\$} \mathbb{Z}_Q$ and $\boldsymbol{p}'_1 \leftarrow \texttt{split}(m'_1)$, to the Client; $[\![\Delta_0 \cdot m'_1]\!]$ and $[\![\Delta_0 \cdot \boldsymbol{p}'_1]\!]$ shared

	Server	**Client**
1	$([\![v]\!]_0, [\![\Delta_0 \cdot [\![v]\!]_1]\!]_0) \leftarrow [\![v]\!\rangle_0$	$([\![v]\!]_1, [\![\Delta_0 \cdot [\![v]\!]_1]\!]_1) \leftarrow [\![v]\!\rangle_1$
2		$[\![v']\!]_1 \leftarrow [\![v]\!]_1 - m'_1$
3	$[\![\Delta_0 \cdot [\![v']\!]_1]\!]_0 \leftarrow [\![\Delta_0 \cdot [\![v]\!]_1]\!]_0 - [\![\Delta_0 \cdot m'_1]\!]_0$	$[\![\Delta_0 \cdot [\![v']\!]_1]\!]_1 \leftarrow [\![\Delta_0 \cdot [\![v]\!]_1]\!]_1 - [\![\Delta_0 \cdot m'_1]\!]_1$
4	Receive $[\![v']\!]_1$, h'	Send $[\![v']\!]_1$, $h([\![\Delta_0 \cdot [\![v']\!]_1]\!]_1)$
5	Verify MAC for $[\![v']\!]_1$, $[\![\Delta_0 \cdot [\![v']\!]_1]\!]_0$, h'	
6	$[\![\boldsymbol{p}]\!]_0 \leftarrow \texttt{split}([\![v]\!]_0)$	
7	$[\![\boldsymbol{p}]\!\rangle_0 \leftarrow ([\![\boldsymbol{p}]\!]_0, [\![\Delta_0 \cdot \boldsymbol{p}'_1]\!]_0)$	$[\![\boldsymbol{p}]\!\rangle_1 \leftarrow (\boldsymbol{p}'_1, [\![\Delta_0 \cdot \boldsymbol{p}'_1]\!]_1)$

The CR for the parallel version of Protocol 8 is generated differently from what is described in Protocol 2, due to the requirement that m'_1 and $\boldsymbol{p}'_1$ should be sent directly to the Client. First, seeds m_s and p_s are obtained for the Client. Then, m_s is expanded into the values m'_1 and $[\![\Delta_0 \cdot m'_1]\!]_1$. After that, $\boldsymbol{p}'_1$ is computed from m'_1. Next, p_s is expanded into $[\![\Delta_0 \cdot \boldsymbol{p}'_1]\!]_1$. Finally, the Server shares of the MACs are computed.

Note that in the protocol the Server can obtain $\langle\!\langle \boldsymbol{p} \rangle\!\rangle_0$ locally, because the Client MACs on $[\![\boldsymbol{p}]\!]$ are not used. Since Client MACs are not used, the protocol is only passively secure against potentially malicious Server. The security implications of this are discussed in Section 5.2.

3.4 Long overflow

This section describes the long overflow protocol and its subprotocol, the short overflow protocol. These protocols are the modified versions of the binary overflow protocols presented in *Attrapadung et. al* [14, Protocols 5 and 6]. In the next Section, the long overflow protocol will be combined with the asymmetric reshare to parts protocol from the previous Section, in order to compute inequalities between shared values and a known constant.

The Protocol 9 (`short_overflow`), takes $[\![v]\!]^{(q)}$, the radix $r \in \mathbb{Z}$ and the modulus q_M to obtain $[\![b]\!]^{(q_M)}$ (is-border) and $[\![c]\!]^{(q_M)}$ (is-carry), such that

$$[\![b]\!]^{(q_M)} = \begin{cases} 1 & \text{if } \widetilde{v} = r - 1 \\ 0 & \text{else} \end{cases} \text{ and } [\![c]\!]^{(q_M)} = \begin{cases} 1 & \text{if } \widetilde{v} \geq r \\ 0 & \text{else} \end{cases}.$$

The condition $q \geq 2 \cdot r$ must hold for the `short_overflow` protocol and, by extension, the `long_overflow` protocol.

Protocol 9: `short_overflow`

Input: $[\![v]\!]^{(q)}$, r, q_M
Output: $[\![b]\!]^{(q_M)}$, $[\![c]\!]^{(q_M)}$

1 $[\![\boldsymbol{w}]\!]^{(q_M)} \leftarrow$ `characteristic_vector`$([\![v]\!]^{(q)}, q_M)$
2 $[\![b]\!]^{(q_M)} \leftarrow [\![w_{r-1}]\!]^{(q_M)}$
3 $[\![c]\!]^{(q_M)} \leftarrow \sum_{i=r}^{q_M-1} [\![w_i]\!]^{(q_M)}$

The Protocol 10 (`long_overflow`) from the radix r, the desired output modulus Q and a list of parts $[\![\boldsymbol{p}]\!]^{(q)}$, such that $\forall_{i=0}^{d-1}\,(\widetilde{p_i} \leq 2 \cdot (r-1))$, obtains $[\![b]\!]^{(Q)}$, where

$$b = \begin{cases} 1 & \text{if } \sum_{i=0}^{|\boldsymbol{p}|-1} r^i \cdot \widetilde{p_i} \geq r^{|\boldsymbol{p}|} \\ 0 & \text{else} \end{cases}.$$

This condition is equivalent to whether a carry (overflow) would occur from the most significant digit (part), if all the carries would be propagated.

Protocol 10: `long_overflow`

Input: $[\![\boldsymbol{p}]\!]^{(q)}$, r, Q
Output: $[\![t]\!]^{(Q)}$

1 $q_M \leftarrow \min\{n \in \mathbb{P} : n > 2^{|p|}\}$ ▷ In Duolithium, $q_M = 67$
2 **for** $i \in \{0, \ldots, |\boldsymbol{p}| - 1\}$ **in parallel:**
3 $\quad \left(\left[\!\left[b_i\right]\!\right]^{(q_M)}, \left[\!\left[c_i\right]\!\right]^{(q_M)}\right) \leftarrow$ `short_overflow`$([\![p_i]\!]^{(q)}, r, q_M)$
4 $[\![m]\!]^{(q_M)} \leftarrow \sum_{i=1}^{|\boldsymbol{r}|-1} 2^{i-1} \cdot [\![b_i]\!]^{(q_M)} + \sum_{i=0}^{|\boldsymbol{r}|-1} 2^i \cdot [\![c_i]\!]^{(q_M)}$
5 $[\![\boldsymbol{k}]\!]^{(Q)} \leftarrow$ `characteristic_vector`$([\![m]\!]^{(q_M)}, Q)$
6 $[\![t]\!]^{(Q)} \leftarrow \sum_{i=2^{|p|-1}}^{q_M} [\![k_i]\!]^{(Q)}$

Both protocols discussed in this section can be trivially extended to support Server MACs (or both MACs).

3.5 Rejection sampling

Rejection sampling (rejection check) is one of the two main operations in ML-DSA signing. The rejection check protocol (Protocol 11), from $[\![\boldsymbol{v}]\!\!]^{(Q)}$ and border values $\boldsymbol{g}$ ($|\boldsymbol{g}| = |\boldsymbol{v}|$), obtains

$$t = \begin{cases} \textit{true} & \text{if } \forall_{i=0}^{|\boldsymbol{v}|-1}\,(|v_i \bmod^{\pm} Q| < g_i) \\ \textit{false} & \text{otherwise} \end{cases}.$$

That is, the return value is *true* if and only if the rejection check passed.

The protocol is actively secure against a corrupted Client and *passively* secure against a corrupted Server. The security implications of this are discussed in Section 5.2.

Protocol 11: `rejection_check`

Input: $[\![\boldsymbol{v}]\!]^{(Q)}$, $\boldsymbol{g}$
Output: t

1 **for** $i \in \{0, \ldots, |\boldsymbol{v}| - 1\}$ **in parallel:**
2 $\quad [\![v'_i]\!]^{(Q)} \leftarrow [\![v_i]\!]^{(Q)} + g_i - 1$
3 $\quad g'_i \leftarrow 2 \cdot g_i - 1$ ▷ $g'_i \in \mathbb{Z}_Q$
4 $\quad [\![\boldsymbol{p}]\!]^{(q)} \leftarrow$ `reshare_to_parts_ap`$([\![v'_i]\!]^{(Q)})$

	Server	**Client**
5	$([\![\boldsymbol{p}]\!]_0^{(q)}, [\![\Delta_0 \cdot [\![\boldsymbol{p}]\!]_1]\!]_0^{(q)}) \leftarrow [\![\boldsymbol{p}]\!]_0^{(q)}$	
6	$w_0 \leftarrow$ `unsplit`$([\![\boldsymbol{p}]\!]_0^{(q)})$	
7	$w'_0 \leftarrow w_0 - g'_i$	
8	$n \leftarrow r^d - Q$	
9	$[\![\boldsymbol{a}]\!]_0^{(q)} \leftarrow (\texttt{split}(\widetilde{w_0} + n), [\![\Delta_0 \cdot [\![\boldsymbol{p}]\!]_1]\!]_0^{(q)})$	$[\![\boldsymbol{a}]\!]_1^{(q)} \leftarrow [\![\boldsymbol{p}]\!]_1^{(q)}$
10	$[\![\boldsymbol{b}]\!]_0^{(q)} \leftarrow (\texttt{split}(\widetilde{w'_0} + n), [\![\Delta_0 \cdot [\![\boldsymbol{p}]\!]_1]\!]_0^{(q)})$	$[\![\boldsymbol{b}]\!]_1^{(q)} \leftarrow [\![\boldsymbol{p}]\!]_1^{(q)}$
11	$[\![c]\!]_0^{(q_r)} \leftarrow \begin{cases} 1 & \text{if } \widetilde{w_0} \geq g'_i \\ 0 & \text{else} \end{cases}$	$[\![c]\!]_1^{(q_r)} \leftarrow 0$

12 $\quad [\![y_a]\!]^{(q_r)} \leftarrow$ `long_overflow`$([\![\boldsymbol{a}]\!]^{(q)}, r, q_r)$
13 $\quad [\![y_b]\!]^{(q_r)} \leftarrow$ `long_overflow`$([\![\boldsymbol{b}]\!]^{(q)}, r, q_r)$ ▷ In parallel with previous
14 $\quad [\![r_i]\!]^{(q_r)} \leftarrow [\![y_b]\!]^{(q_r)} - [\![y_a]\!]^{(q_r)} + [\![c]\!]^{(q_r)}$
15 $[\![k]\!]^{(q_r)} \leftarrow \sum_{i=0}^{|\boldsymbol{v}|-1} [\![r_i]\!]^{(q_r)}$
16 $[\![\boldsymbol{x}]\!]^{(Q)} \leftarrow$ `characteristic_vector`$([\![k]\!]^{(q_r)}, Q)$
17 $[\![y]\!]^{(Q)} \leftarrow [\![x_0]\!]^{(Q)}$
18 Parties declassify y
19 $t \leftarrow \begin{cases} \textit{true} & \text{if } y = 1 \\ \textit{false} & \text{otherwise} \end{cases}$

The constants r and d (on the line 8), should match the values that are used in the `reshare_to_parts_ap` protocol. As mentioned in Section 3.3, in Duolithium: $r = 15$, $d = 6$ and $q = 29$.

Lines 2–4 of Protocol 11 compute new values v'_i and borders g'_i that will be used in the later parts of the computation, such that v_i passes the rejection check if and only if $0 \leq v'_i < g'_i$.

Then, the protocol computes an inequality between $[\![v'_i]\!]$ and the appropriate border g'_i. The procedure for computing the inequality is derived from the comparison protocol as

presented by *Attrapadung et. al* [14, Protocol 7]. This procedure ensures that $r_i = 1$ if the element v_i is outside of the allowed region, and $r_i = 0$ otherwise. Note that in the protocol presented here, the Server uses $[\![\boldsymbol{p}]\!]_0$ to reconstruct the value w_0. This is because the used `reshare_to_parts_ap` protocol effectively reshares its argument before computing the parts.

The lines 15–17 essentially compute a large conjunction. Note that $x_0 = 1$ if and only if $k = 0$, which is true if and only if $\forall_{i=0}^{|\boldsymbol{v}|-1} r_i = 0$ (assuming a sufficiently large q_r — see below). Thus, as required, $y = 1$ if and only if all the elements passed the rejection check.

In Duolithium, the value q_r is fixed as a constant with a relatively small value: $q_r = 71$. Note that in ML-DSA, for each element that is the rejection check is conducted on there is only a small probability of this element being outside of the allowed range. For any i: $r_i = 1$ if v_i is outside of the allowed range, and $r_i = 0$ otherwise. Therefore, while it is possible that $\sum_{i=0}^{|v|-1} \widetilde{r_i} \geq q_r$ (see line 15), the used value q_r ensures that in practice this happens only with negligible probability (less than 2^{-256} for all ML-DSA parameter sets). Using a relatively small value for q_r reduces the volume of data that needs to be transmitted between the Server and the Client during the executions of the `long_overflow` subprotocol.

Note that Protocol 11 is presented in such a form to be more readable. The behaviour of an optimized Duolithium implementation would differ from this form. For example, all the values in ring q_r would not be saved in memory: rather, the corresponding computations would affect a certain counter — a single shared value with Server MAC.

4 Invented protocols

This section documents the protocols that were created as a part of this thesis research.

Note on the protocol optimization. Aside from correctness and security properties, protocols can also be compared according to these efficiency metrics:

1. Total number of rounds (lower is better). Each time a value is sent by a party, the other party receives it after a certain network delay. The Client may be connected via a carrier provider network or Wi-Fi, both of which introduce significant delays in the transmission of any given data object.

2. Information volume (traffic) sent from Server to the Client and from Client to the Server (lower is better). The Client may be connected to a metered network, the network may have low data throughput.

3. Traffic sent from the CRP to the Server (lower is better).

Protocols 14 and 15 were specifically optimized to achieve lower values on these metrics; these protocols, as presented in this work, differ significantly from the initial draft versions of these protocols.

4.1 Short coefficient generation

This Section introduces the CRP-wariness security property. It then describes the protocol that is necessary for generating vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ during the Duolithium key generation. The last part of the Section shows that the described protocol is CRP-wary.

CRP-wariness property

One desirable property of the key generation protocol is that a potentially corrupt CRP does not affect the security strength of the key, even if it provides CR incorrectly. Let *CRP-wariness* be defined as a property of the protocol which is satisfied if and only if for all adversarial CRP, the honest parties, by executing the protocol, regardless of whether the CR provided by the CRP is correct, either (1) with overwhelming probability obtain the output from the correct distribution, or (2) halt the execution of the protocol. In other words, if the CRP cheats, with overwhelming probability the parties will either detect that the CRP provided incorrect correlated randomness (and halt the protocol), or the security strength of the produced material is not compromised.

The protocol

The short coefficient generation protocol (Protocol 12) from the desired q obtains $\langle\!\langle v \rangle\!\rangle^{(Q)}$, where $v \xleftarrow{\$} \{x \in \mathbb{Z} \mid 0 \leq x < q\}$.

Lines 1–10 of the protocol serve only to verify that $[\![\boldsymbol{t}]\!]^{(Q)}$ (received from the CRP) is suitable for the formation of the shared coefficient $[\![s]\!]$. The parties halt the protocol if the security check on line 2 or the security check on line 10 fails. Only after both checks pass, the parties agree on a random shared value $\langle\!\langle k \rangle\!\rangle^{(q)}$. The procedure parties follow next (on lines 13–15) is equivalent to obtaining the characteristic vector of $\langle\!\langle k \rangle\!\rangle^{(q)}$ in ring $\mathbb{Z}_Q$, per Section 2.8.2, and then using this CV to compute the identity of k — with the result also being in ring $\mathbb{Z}_Q$.

Protocol 12: `generate_short`

Input: q

Output: $\langle\!\langle s \rangle\!\rangle^{(Q)}$

CR: $\langle\!\langle r \rangle\!\rangle^{(q)}$ and $\langle\!\langle \boldsymbol{t} \rangle\!\rangle^{(Q)}$, such that $r \xleftarrow{\$} \mathbb{Z}_q$ and t is CV of r

1 $\langle\!\langle z_A \rangle\!\rangle^{(Q)} \leftarrow \left(\sum_{i=0}^{q-1} \langle\!\langle t_i \rangle\!\rangle^{(Q)}\right) - 1$

2 Parties declassify z and verify that $z_A \stackrel{?}{=} 0$

3 $\boldsymbol{w} \leftarrow \{(i,u) \in \mathbb{Z}_q^2 : i < u\}$

4 Parties derive seed $m_s \in \mathbb{Z}_2^{256}$ using some commitment scheme

5 $m \leftarrow f(m_s)$ ▷ $f : \mathbb{Z}_2^{256} \to \mathbb{Z}_Q^{2\times\xi\times|\boldsymbol{w}|}$

6 **for** $h \in [0;\xi)$ **in parallel:**

7 ⎸ **for** $y \in \{0,\ldots,|w|-1\}$ **in parallel:**

8 ⎸⎸ $i,u \leftarrow w_y$

9 ⎸⎸ $\langle\!\langle z_B \rangle\!\rangle^{(Q)} \leftarrow$ `shares_multiplication`$(m_{(0,h,y)} \cdot \langle\!\langle t_i \rangle\!\rangle^{(Q)}, m_{(1,h,y)} \cdot \langle\!\langle t_u \rangle\!\rangle^{(Q)})$

10 ⎸⎸ Parties declassify z_B and verify that $z \stackrel{?}{=} 0$

11 Each party i generates $[\![k]\!]_i^{(q)} \xleftarrow{\$} \mathbb{Z}_q$

12 $\langle\!\langle k \rangle\!\rangle^{(q)} \leftarrow$ `entry`$([\![k]\!]^{(q)})$

13 $\langle\!\langle d \rangle\!\rangle^{(q)} \leftarrow \langle\!\langle k \rangle\!\rangle^{(q)} + \langle\!\langle r \rangle\!\rangle^{(q)}$

14 Parties declassify d

15 $\langle\!\langle s \rangle\!\rangle^{(Q)} \leftarrow \sum_{i=0}^{q-1} i \cdot \langle\!\langle t_{i+d} \rangle\!\rangle^{(Q)}$ ▷ Indexes are *modulo* q

For the parallel execution of the protocol, the zero check protocol from Section 2.8.3 is used, instead of direct declassify. On line 5, f is a cryptographically secure pseudorandom number generator. The constant ζ is a security parameter, which is relevant for the CRP-wariness property of the protocol.

Discussion on the CRP-wariness of Protocol 12

Let's show[10] that Protocol 12 is CRP-wary. Note that the distribution of s matches the desired distribution if $\boldsymbol{t}$ has exactly one non-zero element and this element is 1 — regardless of whether the pair $(r, \boldsymbol{t})$ is consistent. Therefore, it is sufficient to show that $\boldsymbol{t}$ conforms to this description.

Suppose for the sake of contradiction that there is more than one non-zero element in $\boldsymbol{t}$. Therefore, there exists a pair of elements t_i and t_u, such that neither of them is zero. The result of the multiplication protocol (on line 9) for this pair is $(m_i \cdot t_i) \cdot (m_u \cdot t_u) + c - a \cdot b$, where m_i and m_u were agreed upon by the parties for this pair and t_i, t_u, a, b, c are provided by the CRP (see Section 2.8.1). Since the security check on line 10 passed, the result of the multiplication protocol is equal to zero. Therefore, CRP sent the values such that $a \cdot b - c = x = (m_i \cdot t_i) \cdot (m_u \cdot t_u)$. Note that t_i and t_u are non-zero. From this it must follow that the CRP can derive $m_i \cdot m_u = x \cdot t_i^{-1} \cdot t_u^{-1}$. Actually, CRP has no access to $m_i \cdot m_u$ — a product of two pseudorandom elements of $\mathbb{Z}_Q$, generated by the parties — thus the CRP can predict the product with only the probability of guessing. Due to the construction of the protocol, CRP needs to correctly predict ξ independent products for each pair (t_i, t_u), where ξ is the security amplification parameter (see line 5). Depending on the choice of ξ, the probability of correct guessing can be made negligible.

Suppose for the sake of contradiction that there are no non-zero elements. Therefore, the $z_A \neq 0$. Thus, the security check on line 2 failed. Contradiction.

Therefore, with overwhelming probability, there is exactly one non-zero element.

Suppose for the sake of contradiction that this element is not 1. Therefore, $z_A \neq 0$ and the security check on line 2 failed. Therefore there is only one non-zero element in b and this element is 1.

Thus, we have shown that CRP cannot affect the distribution of s, without there being an overwhelming probability of it being caught by the parties.

In Duolithium, ξ is set as a constant: $\xi = 7$. This ensures that the probability of successful cheating is less than 2^{-128} (the protocol is used with the value of $Q = 8380417$).

It can be noted that deriving values for the second security check is relatively costly in terms of performance and network communication, since a total of $\xi \cdot \frac{q \cdot (q-1)}{2}$ multiplications need to be conducted for one generated $\langle\!\langle s \rangle\!\rangle$. Thankfully, the protocol is used only with $q = 3$ and $q = 5$, and only during the key generation (Protocol 16). In the planned operation of Duolithium, the sharded private key of a specific Client is used to create potentially hundreds of signatures. This is to say, the significant performance impact of the security check is tolerated in the context of the planned Duolithium operation.

[10] What follows is an argument; security proofs are not in the scope of this work.

4.2 Zero-equality

The zero-equality protocol, from a shared value $[\![v]\!]^{(Q)}$ and the border value b obtains $[\![z]\!]^{(Q)}$, such that

$$z = \begin{cases} 1 & \text{if } v = 0 \\ 0 & \text{else} \end{cases}.$$

The protocol is functional only if the input condition is met: $v < b$.

Compared to a zero-equality protocol that can be trivially derived from the characteristic vector protocol, the computational complexity and the CRP-to-Server traffic of the protocol discussed in this section grow linearly in b. This makes the protocol discussed herein practical for large modulo rings, such as formed by ML-DSA $Q = 8380417$.

Protocol 13: `eqzero`

Input: $[\![v]\!]^{(Q)}$, b
Output: $[\![z]\!]^{(Q)}$
CR: $[\![m]\!]^{(Q)}$, where $m \xleftarrow{\$} \mathbb{Z}_Q$, $[\![\boldsymbol{t}]\!]^{(q)}$, such that $|\boldsymbol{t}| = b + 1$ and

$$t_i = \begin{cases} 1 & \text{if } \lfloor \frac{m}{a} \rfloor = i \\ 0 & \text{else} \end{cases}, \text{where } a = \left\lfloor \frac{Q}{b} \right\rfloor$$

1 $a \leftarrow \lfloor \frac{Q}{b} \rfloor$
2 $[\![d]\!]^{(Q)} \leftarrow [\![m]\!]^{(Q)} + a \cdot [\![v]\!]^{(Q)}$
3 Parties declassify d
4 $[\![z]\!]^{(Q)} \leftarrow \left[\!\left[t_{\lfloor \frac{d}{a} \rfloor} \right]\!\right]^{(Q)}$

Theorem 1. *Protocol 13 is functional.*

Proof. There are exactly 2 cases:

1. $v = 0$. In this case, $d = m$ and $z = t_{\lfloor \frac{m}{a} \rfloor}$. Due to the correlated randomness construction, $t_{\lfloor \frac{m}{a} \rfloor} = 1$. Thus, $z = 1$.

2. $v \neq 0$. Then $\lfloor \frac{d}{a} \rfloor = \lfloor \frac{m + a \cdot v}{a} \rfloor$. Note that $a \cdot \widetilde{v} \leq a \cdot (b - 1) = a \cdot b - a \leq Q - a < Q$. Therefore, $\widetilde{m} + a \cdot \widetilde{v} < 2 \cdot Q$. There are exactly 2 cases:

   (a) $\widetilde{m} + a \cdot \widetilde{v} < Q$. Then $\lfloor \frac{m + a \cdot v}{a} \rfloor = \lfloor \frac{\widetilde{m} + a \cdot \widetilde{v}}{a} \rfloor = \widetilde{v} + \lfloor \frac{m}{a} \rfloor > \lfloor \frac{m}{a} \rfloor$. Thus, $z \neq t_{\lfloor \frac{m}{a} \rfloor}$, which (due to the CR construction) implies $z = 0$.

   (b) $Q \leq \widetilde{m} + a \cdot \widetilde{v} < 2 \cdot Q$. Then $d = \widetilde{m} + a \cdot \widetilde{v} - Q \leq \widetilde{m} + (Q - a) - Q = \widetilde{m} - a$. Therefore, $0 \leq \lfloor \frac{d}{a} \rfloor \leq \lfloor \frac{\widetilde{m} - a}{a} \rfloor = \lfloor \frac{\widetilde{m}}{a} \rfloor - 1 < \lfloor \frac{m}{a} \rfloor$. Thus, $\lfloor \frac{d}{a} \rfloor \neq \lfloor \frac{m}{a} \rfloor$. Since $t_i = 0$ for any $i \neq \lfloor \frac{m}{a} \rfloor$, $t_{\lfloor \frac{d}{a} \rfloor} = 0$. Thus $z = 0$.

Therefore, the protocol's behaviour is correct in all cases. □

The protocol is trivially extendable to support MACs. This protocol (with Server and Client MACs) is used in the protocol described in Section 4.4, in effect to compute conjunctions of multiple values in $\{0,1\}$, shared in a ring $\mathbb{Z}_Q$.

Protocol generalization. A more general protocol can be described, which would map a shared value $[\![v]\!]^{(q)}$, such that $v < b$, to a value $[\![f(v)]\!]^{(Q)}$, where $f : \mathbb{Z}_q \to \mathbb{Z}_Q$. For such a protocol, the values $[f(0), \ldots, f(b-1)]$ should be assigned to $\boldsymbol{t}$ (by the CRP), depending on whether $m < a \cdot \lfloor \frac{q}{a} \rfloor$, and on the value $\lfloor \frac{m}{a} \rfloor$.

4.3 Symmetric reshare to parts

This Section documents the symmetric reshare to parts protocol — a subprotocol of the high-bits protocol (see Section 4.4).

The symmetric reshare to parts protocol (Protocol 14) allows, from $\langle\!\langle v \rangle\!\rangle^{(Q)}$ and a list of radixes (the partitioning scheme) $\boldsymbol{r}$, to obtain a list of characteristic vectors $[\langle\!\langle \boldsymbol{b}_0 \rangle\!\rangle^{(Q)}, \ldots, \langle\!\langle \boldsymbol{b}_{|\boldsymbol{r}|-1} \rangle\!\rangle^{(Q)}]$, where $\boldsymbol{b}_i$ is a CV (with length r_i) of p_i, such that $\widetilde{[\![v]\!]_0} + \widetilde{[\![v]\!]_1} = \sum_{i=0}^{|\boldsymbol{r}|-1} (\widetilde{p_i} \cdot \prod_{u=0}^{i-1} r_u)$. Note that the selection is unique. The following input requirement must be satisfied: $\prod_{i=0}^{|\boldsymbol{r}|-1} r_i \geq 2 \cdot Q$.

The general idea of the protocol is that the parties first create the characteristic vectors, trusting each-other (lines 1–10), and then verify that the characteristic vectors were created correctly (lines 11–13). The parts are obtained one after another, while propagating the carry, starting from the least significant digit, as it never receives a carry. For each digit position i, the parties appropriately construct the characteristic vector $\boldsymbol{b}_i$ of p_i, from the values provided by the CRP, as well as appropriately select the value c', which corresponds to whether there is a carry. The parties abort the protocol execution if the security check on line 13 fails. This check ensures that the output of the protocol corresponds to $\widetilde{[\![v]\!]_0} + \widetilde{[\![v]\!]_1} \pmod{Q}$.

The protocol uses the `split` local function, that was introduced in Section 3.3.

Henceforth, let us denote $\text{;}_{i=a}^{b} e(i) := (e(a), e(a+1), \ldots, e(b))$, where e is an expression. The empty product is equal to 1.

Protocol 14: `symmetric_reshare_to_parts`

Input: $\langle\!\langle v \rangle\!\rangle^{(Q)}$, $\boldsymbol{r}$

Output: $[\langle\!\langle \boldsymbol{b}_0 \rangle\!\rangle^{(Q)}, \ldots, \langle\!\langle \boldsymbol{b}_{r-1} \rangle\!\rangle^{(Q)}]$

CR: For $i \in [0, |\boldsymbol{r}|)$:

$[\![m_i]\!]^{(2 \cdot r_i)}$, where $m_i \xleftarrow{\$} \mathbb{Z}_{2 \cdot r_i}$,

$\langle\!\langle \boldsymbol{b}'_i \rangle\!\rangle^{(Q)}, |\boldsymbol{b}'_i| = r_i$, where $b'_{(i,u)} = \begin{cases} 1 & \text{if } (u + \widetilde{m_i}) \bmod^+ r_i = 0 \\ 0 & \text{otherwise} \end{cases}$,

$[\![\boldsymbol{c}_i]\!]^{(2 \cdot r_{i+1})}, |\boldsymbol{c}_i| = 2 \cdot r_i$, where $c_{(i,u)} = \begin{cases} 1 & \text{if } (u - \widetilde{m_i}) \bmod^+ (2 \cdot r_i) \geq r_i \\ 0 & \text{otherwise} \end{cases}$

$\bar{\boldsymbol{r}} \leftarrow \bigtriangledown_{i=0}^{|\boldsymbol{r}|-1} \prod_{u=0}^{i-1} r_i$

Party i computes $[\![\boldsymbol{y}]\!]_i \leftarrow \texttt{split}([\![v]\!]_i^{(Q)}, r)$

$[\![c']\!] \leftarrow 0$

for $i \in [0, |\boldsymbol{r}|)$ **consecutively:**

- $[\![k]\!]^{(2 \cdot r_i)} \leftarrow \left([\![y_i]\!] + [\![c']\!]^{(2 \cdot r_i)}\right) \bmod^+ (2 \cdot r_i)$
- $[\![d]\!]^{(2 \cdot r_i)} \leftarrow [\![m_i]\!]^{(2 \cdot r_i)} + [\![k]\!]^{(2 \cdot r_i)}$
- Parties declassify d
- $\langle\!\langle \boldsymbol{b}_i \rangle\!\rangle^{(Q)} \leftarrow \langle\!\langle b'_{(i,0-d)} \rangle\!\rangle, \langle\!\langle b'_{(i,1-d)} \rangle\!\rangle, \ldots, \langle\!\langle b'_{(i,r_i-1-d)} \rangle\!\rangle$ ▷ Indexes are *modulo* r_i
- $\langle\!\langle p_i \rangle\!\rangle^{(Q)} \leftarrow \sum_{u=0}^{r_i-1} (u \cdot \langle\!\langle b_{(i,u)} \rangle\!\rangle^{(Q)})$
- $[\![c']\!] \leftarrow [\![c_{(i,d)}]\!]$

$\langle\!\langle w \rangle\!\rangle^{(Q)} \leftarrow \langle\!\langle v \rangle\!\rangle^{(Q)} - \sum_{i=0}^{|\boldsymbol{r}|-1} \left(\langle\!\langle p_i \rangle\!\rangle^{(Q)} \cdot \bar{r}_i\right)$

Parties declassify w

Parties verify $w \stackrel{?}{=} 0$

Theorem 2. *Protocol 14 is functional.*

Proof. Let's show, inductively, that at the beginning of each *for* loop iteration (and at the loop exit), $\sum_{u=0}^{i-1} (\widetilde{p_u} \cdot \bar{r}_u) + c' \cdot \bar{r}_i = \sum_{u=0}^{i-1} (\widetilde{y_u} \cdot \bar{r}_u)$. As will be shown below, at the loop exit (when $i = |\boldsymbol{r}|$), $c' = 0$.

- Base case. At the start of the for loop, both the left hand side and the right hand side equal 0, the equation holds.
- Induction step. Note that $\widetilde{[\![y_i]\!]_0} + \widetilde{[\![y_i]\!]_1} \leq 2 \cdot (r_i - 1)$ and $c' \leq 1$, thus $\widetilde{[\![y_i]\!]_0} + \widetilde{[\![y_i]\!]_1} + c' < 2 \cdot r_i$.

  1. Let's show that at the end of the *for* loop iteration $p_i = k \bmod^+ r_i$. Note that $\boldsymbol{b}_i$ is a rearrangement of elements of the characteristic vector $\boldsymbol{b}'_i$ and that

$p_i = \sum_{u=0}^{r_i-1}(u \cdot b_{(i,u)})$. Therefore, $\boldsymbol{b}_i$ is a characteristic vector of p_i. Thus, it is sufficient to show that $b_{(i,k \bmod^+ r_i)} = 1$. This is indeed so:

$$b_{(i,k \bmod^+ r_i)} = b'_{(i,(k-d) \bmod^+ r_i)} = b'_{(i,(k-m_i-k) \bmod^+ r_i)} = b'_{(i,-m_i \bmod^+ r_i)} = 1.$$

2. Let's show that at the end of the *for* loop iteration $c' = \left\lfloor \frac{k}{r_i} \right\rfloor$. Note that $c' = c_{(i,d)} = c_{(i,m_i+k)}$. Due to $\boldsymbol{c}_i$ construction,

$$c_{(i,m_i+k)} = \begin{cases} 1 & \text{if } k \geq r_i \pmod{2 \cdot r_i} \\ 0 & \text{otherwise} \end{cases},$$

which is correct.

It is easy to see that from the previous two points it follows that the equation holds at the start of the next iteration (or at the loop exit if it was the last iteration).

Note that the input condition $\prod_{i=0}^{|\boldsymbol{r}|-1} r_i \geq 2 \cdot Q$ ensures that c' after the last iteration is equal to 0, thus $\sum_{i=0}^{|\boldsymbol{r}|-1}(\widetilde{p_i} \cdot \bar{r}_i) = \sum_{i=0}^{|\boldsymbol{r}|-1}(\widetilde{y_i} \cdot \bar{r}_i)$. At the same time, due to the definition of the `split` routine, $\widetilde{[\![v]\!]_0} + \widetilde{[\![v]\!]_1} = \sum_{i=0}^{|\boldsymbol{r}|-1}(\widetilde{[\![y_i]\!]_0} \cdot \bar{r}_i + \widetilde{[\![y_i]\!]_1} \cdot \bar{r}_i)$. For each i: $\boldsymbol{b}_i$ is a characteristic vector of p_i. The security check on line 13 passes; the protocol is functional. □

For the parallel execution of `symmetric_reshare_to_parts`, the protocol from see Section 2.8.3 is used for the security check, instead of direct declassify.

Note that while this protocol obtains shares with Server MAC and Client MAC, it is not actively secure as-is. The reason is that a potentially malicious party can with non-negligible probability succeed in modifying the output in a way that instead of corresponding to $t = \widetilde{[\![v]\!]_0} + \widetilde{[\![v]\!]_1}$, it would correspond to $t' = k \cdot Q + \widetilde{[\![v]\!]_0} + \widetilde{[\![v]\!]_1}$, where $t' \geq 0$ and $k \neq 0$. Section 4.4 describes a workaround that allows preserving the active security property of the superprotocol despite this trait of this protocol.

4.4 High-bits

Obtaining high bits is one of the two main operations in ML-DSA signing. The high bits protocol (Protocol 15), from $\langle\!\langle v \rangle\!\rangle^{(Q)}$ and α, obtains $\langle\!\langle h \rangle\!\rangle^{(Q)}$, such that

$$h = \begin{cases} \left\lfloor \frac{(v + \frac{\alpha}{2} - 1) \bmod^+ Q}{\alpha} \right\rfloor & \text{if } v \neq Q - \frac{\alpha}{2} \\ 0 & \text{otherwise} \end{cases}.$$

Here, $Q = 8380417$. As can be verified, evaluating the expression above is equivalent to performing the HighBits operation (discussed in Section 2.3). For ML-DSA-44, $\alpha = 190464$, for ML-DSA-65 and ML-DSA-87, $\alpha = 523776$.

Please note the following about the concrete values of Q and α.

For $\alpha = 190464$: $Q - 1 = 44 \cdot \alpha = 44 \cdot (2^{11} \cdot 3 \cdot 31)$

For $\alpha = 523776$: $Q - 1 = 16 \cdot \alpha = 16 \cdot (2^9 \cdot 3 \cdot 11 \cdot 31)$.

These factorizations were used to derive the partition schemes, that are shown on the algorithm line 3. [11]

Protocol 15: `high_bits`

Input: $\langle\!\langle v \rangle\!\rangle^{(Q)}$, α

Output: $\langle\!\langle h \rangle\!\rangle^{(Q)}$

1 $\langle\!\langle v' \rangle\!\rangle \leftarrow \langle\!\langle v \rangle\!\rangle + \frac{\alpha}{2} - 1$

2 $j = \frac{Q-1}{\alpha}$

3 $\boldsymbol{r} \leftarrow \begin{cases} (31, 24, 16, 16, 2 \cdot j + 1) & \text{if } \alpha = 190464 \\ (31, 33, 32, 16, 2 \cdot j + 1) & \text{if } \alpha = 523776 \end{cases}$

4 $[\langle\!\langle \boldsymbol{b}_0 \rangle\!\rangle^{(Q)}, \ldots, \langle\!\langle \boldsymbol{b}_{|\boldsymbol{r}|-1} \rangle\!\rangle^{(Q)}] \leftarrow$ `symmetric_reshare_to_parts`$(\langle\!\langle v' \rangle\!\rangle, \boldsymbol{r})$

5 $\langle\!\langle k_v \rangle\!\rangle \leftarrow |\boldsymbol{r}| - \sum_{i=0}^{|\boldsymbol{r}|-2} \langle\!\langle b_{(i,0)} \rangle\!\rangle - \sum_{i=j+1}^{2 \cdot j} \langle\!\langle b_{(|\boldsymbol{r}|-1,i)} \rangle\!\rangle$

6 $\langle\!\langle k_r \rangle\!\rangle \leftarrow$ `eqzero`$(\langle\!\langle k_v \rangle\!\rangle, |\boldsymbol{r}| + 1)$

7 $\langle\!\langle c_v \rangle\!\rangle \leftarrow 1 - \langle\!\langle b_{(|\boldsymbol{r}|-1,2 \cdot j)} \rangle\!\rangle + \langle\!\langle k_r \rangle\!\rangle$

8 $\langle\!\langle c_r \rangle\!\rangle \leftarrow$ `eqzero`$(\langle\!\langle c_v \rangle\!\rangle, 3)$

9 Parties declassify c_r

10 Parties verify $c_r \stackrel{?}{=} 0$

11 $\langle\!\langle h' \rangle\!\rangle \leftarrow \sum_{i=0}^{j-1} (i \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,i)} \rangle\!\rangle) + \sum_{i=0}^{j-1} (i \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,j+i)} \rangle\!\rangle) + j \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,2 \cdot j)} \rangle\!\rangle$

12 $\langle\!\langle h \rangle\!\rangle \leftarrow \langle\!\langle h' \rangle\!\rangle - \langle\!\langle k_r \rangle\!\rangle$

The Protocol 15 is engineered as follows. First, value v' is computed from v, such that the special case ($v = Q - \frac{\alpha}{2}$) corresponds to the value of $v' = Q - 1$. Then, v' is split into parts in such a way that discrimination on the value of the most significant part $p_{|\boldsymbol{r}|-1}$ would allow obtaining the correct output value. The discrimination is performed on the line 11, where the coefficients for the values in $\boldsymbol{b}_{|\boldsymbol{r}|-1}$ are selected precisely in the way that would obtain the correct output, with the exception of the following unfortunate case: if $w = \widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1} \geq Q$ and $w - (Q - 1) = 0 \pmod{\alpha}$, the value h' is larger by 1 than the correct output. [12] This is corrected in the next line by subtracting k_r, which is 1 if and only if both of these conditions are true, and 0 otherwise[13]. This value is obtained

[11] It is a very fortunate happenstance that values α do not have large factors. If α would have had a large factor, the described high-bits protocol would not have been practical due to the computational cost and CRP-to-Server traffic in the `symmetric_reshare_to_parts` subprotocol.

[12] Surprisingly, the value $w = Q - 1$ does not require special handling, given the values of the coefficients on line 11.

[13] As a protocol design decision, it was decided that the correction should also be applied to the value $w = 2 \cdot Q - 2$. Thus, a suitable coefficient for $b_{(|\boldsymbol{r}|-1,2 \cdot j)}$ was specified on line 11 and an upper bound for the second sum on line 5 was adjusted. This allows reusing k_v for the security check, which is an optimization over an earlier design of the same protocol.

(on lines 5–6), effectively by performing a conjunction of 5 values. The first four values correspond to the other parts, whereas a value is 1 in case if the corresponding part has a value of 0. The conjunction of these four values gives 1 if and only if $w \bmod^{+} (Q-1) = 0$.

The fifth value is obtained from $\boldsymbol{b}_{|\boldsymbol{r}|-1}$ such that the value of k_r would be 1 only if $(Q-1)+\alpha \le w \le 2\cdot(Q-1)$. If $Q-1 \le w < (Q-1)+\alpha$, the conjunction of 5 values will lead to 0, as required.

Lines 7–10 act as a security check, the purpose of which will be described in a later part of this Section.

For the parallel execution of the protocol, the traffic between the Server and the Client may be reduced if the parties only declassify the sum of the c_r values[14].

Computational performance of the `high_bits` and `symmetric_reshare_to_parts` protocol pair can be improved by implementing certain joint optimizations. For example, computing the complete characteristic vectors for the digits other than the most significant one is not necessary.

Theorem 3. *The Protocol 15 is functional.*

Proof. Note that the protocol is functional if

$$h = \begin{cases} \lfloor \frac{v'}{\alpha} \rfloor & \text{if } v' \neq Q-1 \\ 0 & \text{otherwise} \end{cases}.$$

Let's consider the possible values of $\boldsymbol{p} = \,,_{i=0}^{|\boldsymbol{r}|-1} \sum_{u=0}^{r_i} \left(u \cdot b_{(i,u)}\right)$. There are exactly three cases (note that the security check is not triggered in any of these):

1. $0 \le p_{|\boldsymbol{r}|-1} < j$. In this case, $w = \widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1} < Q-1$. The values during the execution of the protocol will be: $k_v \ge 1$ and $k_r = 0$; $h = \sum_{i=0}^{j-1} \left(i \cdot b_{(|\boldsymbol{r}|-1,i)}\right) + 0 + 0 - 0 = p_{|\boldsymbol{r}|-1}$. Note that $\alpha = \prod_{i=0}^{|\boldsymbol{r}|-2}$. Thus, $h = p_{|\boldsymbol{r}|-1} = \left\lfloor \frac{v'}{\alpha} \right\rfloor$, which is correct, since $v' < Q-1$ (follows from $0 \le p_{|\boldsymbol{r}|-1} < j$).

2. $j \le p_{|\boldsymbol{r}|-1} < 2\cdot j$. In this case $Q-1 \le w < 2\cdot Q-2$.

   Firstly, consider the case where $v' = Q-1$. Then, $p_{|\boldsymbol{r}|-1} = j$ and $\forall_{i=0}^{|\boldsymbol{r}|-2} p_i = 0$. This leads to $k_r = 0$ and $h = 0 + \sum_{i=0}^{j-1} \left(i \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,j+i)} \rangle\!\rangle\right) + 0 - 0 = 0$, which is correct.

   Secondly, consider the case where $v' \neq Q-1$. Note that

   $$\left\lfloor \frac{v'}{\alpha} \right\rfloor = \left\lfloor \frac{w-Q}{\alpha} \right\rfloor = \left\lfloor \frac{w-1}{\alpha} \right\rfloor - j = \begin{cases} \lfloor \frac{w}{\alpha} \rfloor - j & \text{if } w-(Q-1) \neq 0 \mod \alpha \\ \lfloor \frac{w}{\alpha} \rfloor - j - 1 & \text{otherwise} \end{cases}.$$

   (a) If $w-(Q-1) \neq 0 \pmod{\alpha}$: $\exists_{i=0}^{|\boldsymbol{r}|-2} p_i \neq 0$. Thus, $k_v \neq 0$ and $k_r = 0$. Therefore, $h = 0 + \sum_{i=0}^{j-1} \left(i \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,j+i)} \rangle\!\rangle\right) + 0 - 0 = p_{|\boldsymbol{r}|-1} - j$.

[14] This assumes that there are fewer than Q values.

(b) If $w-(Q-1)=0 \pmod{\alpha}$: $\forall_{i=0}^{|\boldsymbol{r}|-2} p_i = 0$. Thus, $k_v = 0$ and $k_r = 1$. Therefore, $h = 0 + \sum_{i=0}^{j-1}(i \cdot \langle\!\langle b_{(|\boldsymbol{r}|-1,j+i)} \rangle\!\rangle) + 0 - 1 = p_{|\boldsymbol{r}|-1} - j - 1$.

Note that $\lfloor \frac{w}{\alpha} \rfloor = p_{|\boldsymbol{r}|-1}$. Therefore, for both (a) and (b): $h = \lfloor \frac{v'}{\alpha} \rfloor$, which is correct.

3. $p_{|\boldsymbol{r}|-1} = 2 \cdot j$. During the protocol execution by honest parties[15], this equality holds only if $\widetilde{[\![v']\!]_0} = \widetilde{[\![v']\!]_1} = Q-1$, which implies $w = 2 \cdot Q - 2$. For $w = 2 \cdot Q - 2$, values during the execution of the protocol will be: $k_v = 0$, $k_r = 1$ and $h = 0+0+j-1$. This is correct, since $\widetilde{[\![v']\!]_0} = \widetilde{[\![v']\!]_1} = Q-1$ implies $v' = Q-2$, and $\lfloor \frac{v'}{\alpha} \rfloor = \lfloor \frac{Q-2}{\alpha} \rfloor = j-1$.

Therefore, the protocol is functional. □

As discussed in the previous Section, the `symmetric_reshare_to_parts` protocol is not actively secure as-is, since if an active attack is performed, it may return characteristic vectors for parts that correspond to the value $t' = k \cdot Q + \widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1}$, $t' \geq 0$, $k \neq 0$. Suppose that such an attack was conducted successfully. Let's consider all the possible cases:

1. $0 \leq t' \leq Q-1$ and $t' = t - Q$, or $Q \leq t' \leq 2 \cdot Q - 2$ and $t' = t + Q$. As shown above, the protocol is functional for all values $0 \leq w \leq 2 \cdot Q - 2$: the protocol returns the same value $[\![h]\!]$ for the shared input $[\![v']\!]$, regardless whether $w < Q$ or $w \geq Q$. Therefore, even if the attack on `symmetric_reshare_to_parts` is successful, `high_bits` produces the correct output.

2. $t' > 2 \cdot Q - 2$. Note that, due to the specific choice of values $\boldsymbol{r}$: $b_{(|\boldsymbol{r}|-1,2\cdot j)} = 1$ and $\exists_{i=0}^{|\boldsymbol{r}|-1-1} b_{(i,0)} = 1$. Therefore, during the execution of the protocol, $k_v \neq 0$, from which $k_r = 0$. Following this, $c_v = 0$ and $c_r = 1$. The security check on line 10 will fail — the attack is detected, execution of the protocol is stopped and the secret material, necessary for creating signatures, is erased.

Note that the other cases are not possible due to the specific choice of values $\boldsymbol{r}$. Therefore, despite the limitation of the `symmetric_reshare_to_parts` protocol, a potential attacker cannot force the `high_bits` protocol to output incorrect values. As will be discussed in the following subsection, a potential attacker's ability to learn information about the protocol inputs is drastically limited.

[15]The argument for the active security property of this protocol follows the functionality proof.

Selective failure attack

Very generally, a selective failure attack is a type of attack where the adversary changes their behaviour during the protocol execution in a way that would trigger or not trigger a failure (protocol abort), depending on the input value of another party. As one example: in a Yao garbled circuit construction, the circuit constructor party can create the circuit in such a way, that evaluating it would fail on a certain input of the evaluator party [15, Section 3]. If the circuit constructor party learns, whether the protocol abort occurs, it can learn information about the input of the evaluator party.

As discussed above, the `symmetric_reshare_to_parts` protocol is not secure against an active attack, where the attacker may cause it to return characteristic vectors for parts that correspond to the value $t' = k \cdot Q + \widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1}$, $t' \geq 0$, $k \neq 0$. For a specific k, whether the attack is detected depends on the value of $\widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1}$. Therefore, the `high_bits` protocol has the unfortunate property of allowing a selective failure attack.

Such an attack could be conducted as follows. After the computation of $\langle\!\langle v' \rangle\!\rangle$, the attacker would make a guess of whether $\widetilde{[\![v']\!]_0} + \widetilde{[\![v']\!]_1} = w < Q$. Then, the attacker would influence the execution of the `symmetric_reshare_to_parts` protocol, so, depending on the guess, parts for $w + Q$ (if the guess is $w < Q$) or $w - Q$ (if the guess is $w \geq Q$) would be obtained. Note that the malicious party will be caught if the guess was incorrect. In this case, the attacker would not learn anything that depends on the private key; the honest party will erase its private key shares. However, if the malicious party is not caught, it learns some information about the distribution of v, which, stemming from the construction of the superprotocol[16], may be useful for deriving information about the secret material.

Despite that, for every useful guess, there is always a risk that the guess will be incorrect and the attack will be detected. Any adversary, on average, would be able to obtain no more than 1 bit of information about the secret material before it is caught. The probability that an adversary would obtain k or more bits of information is at most 2^{-k}. The detection of an attack erases a private key shard and a new private key should be generated to continue using the system. In the planned Duolithium operation, this is a manual process.

Considering the above, we believe that this attack does not constitute a practical security risk.

[16]The only superprotocol of high bits protocol is `signing_attempt`, which will be discussed in Section 5.2.

5 Principal 2PC ML-DSA protocols

The protocols listed in this chapter allow for the ML-DSA key generation and signature creation in the two-party (plus CRP) setting. For details on the construction of the original ML-DSA algorithms, please refer to Sections 2.2–2.4. The two protocols discussed in this Chapter rely on the protocols described in the previous chapters of this thesis and are intended to be the "entry points" of Duolithium.

In the protocols of this chapter, $n, l, k, \eta, \gamma_1, \gamma_2, \beta$ are compile-time integer constants defined by the used ML-DSA parameter set (the concrete values are provided in Table 1).

Both protocols may run only after the Δ values are communicated by the parties to the CRP. In the case of key generation, these values are freshly generated by the parties. In the case of signing, the parties reuse the Δ values that they have memorized after the key generation process has finished.

5.1 Key generation

The Protocol 16 describes the key generation process in Duolithium. After successfully executing this protocol, both parties save the public key (in the ML-DSA format), the complementary private key shards, as well as the Δ values that are necessary for the formation of MACs for future executions of the signature creation protocol.

Protocol 16: keygen

Input: $\varnothing$

Output: Public key, sharded private key

$(q_s, y_s) \leftarrow \begin{cases} (5,1) & \text{if } \eta = 2 \\ (3,2) & \text{if } \eta = 4 \end{cases}$

for $i \in [0; l+k)$ **in parallel:**
  for $u \in [0; n)$ **in parallel:**
    for $j \in [0; y_s)$ **in parallel:**
      $\langle\!\langle c_j \rangle\!\rangle \leftarrow$ generate_short(q_s)
    $\langle\!\langle r_{(i,u)} \rangle\!\rangle \leftarrow \left(\sum_{j=0}^{y_s-1} (q_s)^j \cdot \langle\!\langle c_j \rangle\!\rangle\right) - \eta$

$\langle\!\langle \boldsymbol{s}_1 \rangle\!\rangle \leftarrow \langle\!\langle r_0 \rangle\!\rangle, \langle\!\langle r_1 \rangle\!\rangle, \ldots, \langle\!\langle r_{l-1} \rangle\!\rangle$

$\langle\!\langle \boldsymbol{s}_2 \rangle\!\rangle \leftarrow \langle\!\langle r_l \rangle\!\rangle, \langle\!\langle r_{l+1} \rangle\!\rangle, \ldots, \langle\!\langle r_{l+k-1} \rangle\!\rangle$

Parties derive seed ρ using some commitment scheme

Parties expand ρ into A

$\langle\!\langle \boldsymbol{t} \rangle\!\rangle \leftarrow A\langle\!\langle \boldsymbol{s}_1 \rangle\!\rangle + \langle\!\langle \boldsymbol{s}_2 \rangle\!\rangle$ ▷ NTT forms are used for matrix multiplication

Parties declassify $\boldsymbol{t}$

Parties compute the public key from $(\rho, \boldsymbol{t})$

Parties compute the public key hash tr

Party i saves the private key shard as $(\rho, tr, \langle\!\langle \boldsymbol{s}_1 \rangle\!\rangle_i, \langle\!\langle \boldsymbol{s}_2 \rangle\!\rangle_i, \boldsymbol{t})$

The coefficients of $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ should be sampled from the distribution $\{x : |x| \leq \eta\}$.

- For ML-DSA-44 and ML-DSA-87, $\eta = 2$, which corresponds to $2 \cdot 2 + 1 = 5$ possible coefficient values. Each coefficient for s_1 and s_2 is obtained (independently) as $\langle\!\langle s \rangle\!\rangle = \langle\!\langle s_A \rangle\!\rangle - 2$, where $\langle\!\langle s_A \rangle\!\rangle$ is generated by the `generate_short` protocol with $q = 5$.
- For ML-DSA-65, $\eta = 4$, which corresponds to $2 \cdot 4 + 1 = 9$ possible coefficient values. Coefficients are obtained (independently) as $\langle\!\langle s \rangle\!\rangle = 3 \cdot \langle\!\langle s_A \rangle\!\rangle + \langle\!\langle s_B \rangle\!\rangle - 4$, where $\langle\!\langle s_A \rangle\!\rangle$ and $\langle\!\langle s_B \rangle\!\rangle$ are generated (independently) by the `generate_short` protocol with $q = 3$.

The behaviour of Protocol 16 deviates from the key generation algorithm behaviour as specified in FIPS 204. Specifically, Protocol 16 effectively generates $\boldsymbol{s}_1, \boldsymbol{s}_2$ coefficients uniformly at random (from the allowed range), instead of deriving the coefficients from the seed ρ'. Also, the seed ρ is generated directly, instead of being derived from ζ. Neither of these deviations affect the security of the key generation, because for the security proofs of ML-DSA, the values $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ and ρ are considered to be sampled uniformly at random from their respective distributions. The value K is omitted entirely, since it is not used for deriving the masking vector during the signing (see next Section). Additionally, the entire value $\boldsymbol{t}$ is saved as a part of the private key, since it is needed during the signing: for the local rejection check and for producing hints.

5.2 Signing

The Protocol 17 describes the signing in Duolithium. It relies on Protocol 18 to perform signing attempts.

Please note the following peculiarities of the protocol:

- The Client derives μ locally and then sends it to the Server. Thus, the Server never receives M and it cannot verify whether μ is derived correctly or is malformed.
- Only the Client learns the signature value, since the Server never learns $\langle\!\langle \boldsymbol{z} \rangle\!\rangle_1$.
- After assembling $\boldsymbol{z}$, the Client performs the `local_rejection_check` procedure.
- The Client performs `make_hint` locally and notifies the Server on whether it was successful. The Server cannot verify whether the reported and the actual outcome match.

Protocol 17: `sign`

Input: Message M (on the Client side), private key shards
Output: Signature (on the Client side)

Party i loads private key shard $(\rho, tr, \langle\langle \boldsymbol{s}_1 \rangle\rangle_i, \langle\langle \boldsymbol{s}_2 \rangle\rangle_i, \boldsymbol{t})$
Client derives μ (from public key hash tr and M) and sends it to the Server
Parties expand ρ into A
for $i \in \{0, \dots\}$ **consecutively:**
  $r \leftarrow$ `signing_attempt`$(\boldsymbol{s}_1, \boldsymbol{s}_2, A, \mu)$
  if $r = \perp$ **then**
    continue
  $\langle\langle \boldsymbol{z} \rangle\rangle, c_t \leftarrow r$

Server	**Client**
Reveals $\langle\langle \boldsymbol{z} \rangle\rangle_0$	Accepts $\langle\langle \boldsymbol{z} \rangle\rangle_0$ and verifies MAC
	$\boldsymbol{z} \leftarrow \langle\langle \boldsymbol{z} \rangle\rangle_0 + \langle\langle \boldsymbol{z} \rangle\rangle_1$
	Performs `local_rejection_check`
	Performs `make_hint`, obtaining h

  Client notifies the Server whether $h = \perp$
  if $h = \perp$ **then**
    continue
  else
    Client computes the signature from $(\boldsymbol{z}, c_t, h)$

Note the following way in which the `rejection_check` performed in Protocol 18 differs from the rejection sampling operation performed by the standardized ML-DSA signing algorithm. The standardized algorithm verifies that all coefficients of $\boldsymbol{r}'^L = (\boldsymbol{w} - c \cdot \boldsymbol{s}_2)^L$ fall into the desired region, whereas Protocol 18 performs this check (with the same region) on the value $\boldsymbol{r}^L = \boldsymbol{w}^L - c \cdot \boldsymbol{s}_2$. Note, however, that a coefficient of $\boldsymbol{r}'^L$ falls within the desired region if and only if the corresponding coefficient of $\boldsymbol{r}^L$ falls within this region. Thus, the `rejection_check` in Protocol 18 and the rejection sampling operation in the standardized ML-DSA signing algorithm are equivalent.

After the Server reveals $\langle\langle \boldsymbol{z} \rangle\rangle_0$ to the Client, the Client performs an additional `local_rejection_check` with a purpose that will be made clear later in this Section. Firstly, this check verifies that for every coefficient c of $\boldsymbol{z}$: $|c \bmod^{\pm} q| < \gamma_1 - \beta$. This part matches the first rejection sampling condition of the standardized ML-DSA signing algorithm. Secondly, this check computes $\boldsymbol{r}''^L = (A\boldsymbol{z} - c \cdot \boldsymbol{t})$ and verifies that for every coefficient c of $\boldsymbol{r}''^L$: $|c \bmod^{\pm} q| < \gamma_2 - \beta$. However, it is a known fact [6, Section 1.1, Verification] that $A\boldsymbol{z} - c \cdot \boldsymbol{t} = A\boldsymbol{y} - c \cdot \boldsymbol{s}_2$. Thus (since $A\boldsymbol{y} = \boldsymbol{w}$), $\boldsymbol{r}'^L = \boldsymbol{r}''^L$, which is to say that this part is equivalent to the second rejection sampling condition. Therefore, the `local_rejection_check` is equivalent to the `rejection_check`, but can be performed by the Client that possesses $\boldsymbol{z}$, without possessing the private key.

Protocol 18: `signing_attempt`

Input: $\boldsymbol{s}_1, \boldsymbol{s}_2, A, \mu$

Output: r, such that either $r = (\langle\!\langle \boldsymbol{z} \rangle\!\rangle, c_t)$, or $r = \perp$

for $i \in [0; l)$ **in parallel:**
  for $u \in [0; 256)$ **in parallel:**
    $\langle\!\langle y_{(i,u)} \rangle\!\rangle \leftarrow$ `generate_y`$(\log_2 \gamma_1)$ ▷ γ_1 is a power of 2

$\langle\!\langle \boldsymbol{w} \rangle\!\rangle \leftarrow A\langle\!\langle \boldsymbol{y} \rangle\!\rangle$ ▷ NTT forms are used for matrix multiplication

$\langle\!\langle \boldsymbol{w}^H \rangle\!\rangle \leftarrow$ `high_bits`$(\langle\!\langle \boldsymbol{w} \rangle\!\rangle, 2 \cdot \gamma_2)$

Parties declassify $\boldsymbol{w}^H$

Parties compute seed c_t from $\mu, \boldsymbol{w}^H$

Parties compute c from c_t

$\langle\!\langle \boldsymbol{z} \rangle\!\rangle \leftarrow \langle\!\langle \boldsymbol{y} \rangle\!\rangle + c \cdot \langle\!\langle \boldsymbol{s}_1 \rangle\!\rangle$ ▷ NTT forms are used for multiplication

$\langle\!\langle \boldsymbol{w}^L \rangle\!\rangle \leftarrow \langle\!\langle \boldsymbol{w} \rangle\!\rangle - 2\gamma_2 \cdot \langle\!\langle \boldsymbol{w}^H \rangle\!\rangle$

$\langle\!\langle \boldsymbol{r}^L \rangle\!\rangle \leftarrow \langle\!\langle \boldsymbol{w}^L \rangle\!\rangle - c \cdot \langle\!\langle \boldsymbol{s}_2 \rangle\!\rangle$ ▷ NTT forms are used for multiplication

$\boldsymbol{b} \leftarrow \left(\bigtriangledown_{i=0}^{|\boldsymbol{z}|-1}(\gamma_1 - \beta) \mid \bigtriangledown_{i=0}^{|\boldsymbol{r}^L|-1}(\gamma_2 - \beta) \right)$

$h \leftarrow$ `rejection_check`$(\langle\!\langle \boldsymbol{z} \rangle\!\rangle | \langle\!\langle \boldsymbol{r}^L \rangle\!\rangle, \boldsymbol{b})$

if h **then**
  $r \leftarrow (\langle\!\langle \boldsymbol{z} \rangle\!\rangle, c_t)$

else
  $r \leftarrow \perp$

Remember that the `rejection_check` protocol is only *passively* secure against a potentially malicious Server. That is, a malicious Server can change its share or shares and thus alter (interfere with) the output of the protocol. Consider the following scenarios.

- Suppose the expected result is *true*, but the malicious Server affected the execution such that the actual result was *false*. In this case, the signing attempt fails and the Client will conduct the next signing attempt. This merely gives the Server an opportunity to stall the signing until the Client gives up.
- Suppose the malicious Server caused the result to be *true*. In this case, the honest Client expects to receive $\langle\!\langle \boldsymbol{z} \rangle\!\rangle_0$ and will terminate protocol execution if it does not. As a result of the `local_rejection_check` (which is mathematically equivalent to `rejection_check`), the Client will detect that the Server cheated, since `rejection_check` should have returned *false*. The Client will discard the rejected signature and delete the secret key material.

Therefore, in the context of how the rejection check protocol is used, the lack of active security against a potentially malicious Server (while the passive security is maintained) in this protocol is not a security concern.

Note that the value $\boldsymbol{w}^H$ is declassified during the protocol execution. We believe that this is also not a security concern. Briefly:

- The value $\boldsymbol{w}^H$ itself does not depend on $(\boldsymbol{s}_1, \boldsymbol{s}_2)$, so it initially can be revealed.
- Value $\boldsymbol{w}$ depends on $\boldsymbol{y}$, but $\boldsymbol{w}^H$ does not computably depend on $\boldsymbol{y}$. Thus, even though $\boldsymbol{z}$ and $\boldsymbol{r}^L$ depend on $(\boldsymbol{s}_1, \boldsymbol{s}_2)$ and $\boldsymbol{y}$, revealing $\boldsymbol{w}^H$ alongside (before) h does not reveal more information about $(\boldsymbol{s}_1, \boldsymbol{s}_2)$ and $\boldsymbol{y}$ than revealing h alone.
- If the rejection check passes, ML-DSA hardness assumptions imply that the signature is safe to reveal. Since $\boldsymbol{w}^H$ can be obtained from the public key and the signature with negligible computational effort, it is safe to reveal $\boldsymbol{w}^H$ alongside (before) $\boldsymbol{z}$.

Unlike the ML-DSA signing algorithm, Protocol 18 generates the masking vector $\boldsymbol{y}$ directly. This is the reason why the Duolithium key generation protocol (see the previous Section) does not generate the value K (which is used in the ML-DSA key generation protocol). Again, this does not undermine the security properties of the scheme, because for the security proofs of ML-DSA, the value $\boldsymbol{y}$ is considered to be sampled uniformly at random from the necessary distribution.

The local procedure `make_hint` that is performed by the Client also differs from the standardized version. In Duolithium, the hints are obtained as the positions where the coefficients of $(A\boldsymbol{z} - c \cdot \boldsymbol{t})^H$ and $(A\boldsymbol{z} - c \cdot \boldsymbol{t} + c \cdot \boldsymbol{t}_0)^H$ do not match. Here, $(\cdot)^H$ denotes performing the HighBith operation; $\boldsymbol{t}_0$ is derived from $\boldsymbol{t}$ according to the Power2Round algorithm of the ML-DSA standard. This `make_hint` procedure is equivalent to the standardized make hint algorithm. Note that it uses $\boldsymbol{t}$, which is saved in full during the key generation protocol.

Note that in Protocol 18, μ — the only value that depends on the message — is used for the first time on line 7. Therefore, the signing process may be reorganized such that the `high_bits` protocol and the $\boldsymbol{w}^H$ declassification are performed as a precomputation, perhaps in parallel with the execution of `rejection_check` from a previous signing attempt, or perhaps during times of lesser network and server load (such as at night). Also, multiple signing attempts could be conducted in parallel.

6 Prototype

As a part of this research, a Python implementation of the Duolithium scheme was implemented. This chapter documents this prototype.

6.1 Overview

The prototype is implemented to support key generation and signing, as per the protocols described in the previous chapters of this thesis. All the three standardized ML-DSA parameter sets are supported (ML-DSA-44, ML-DSA-65 and ML-DSA-87)[17].

The prototype is written from scratch, with the exception of some ML-DSA algorithms (as described in FIPS 204), borrowed from the `GiacomoPope/dilithium-py` repository[18].

The prototype consists of 3089 lines of code, of which 763 lines are tests and 266 lines are borrowed (with some adaptations applied) from the repository mentioned above.

The created prototype produces public keys and signatures, whose format matches exactly such of the public keys and signatures created by the ML-DSA algorithms. This means that the keys and signatures created by the prototype can be verified with off-the-shelf cryptographic libraries, such as `liboqs`[19], `Botan3`[20] and others.

A test of prototype functionality was conducted in the following way. First, the CRP and the Server components were launched. Then, for each out of the three ML-DSA parameter sets, 1000 testing iterations were conducted. In each: first the Client component was launched with the key generation command, then launched again to produce one signature of a certain message. All the signatures with their correspondent keys were successfully verified using `Botan3`.

The prototype follows the protocols as presented in this thesis: the sent messages and the performed computations are equivalent to such defined in the protocols. However, to improve computational performance, some operations in the prototype are paralleled, consolidated (fused together) or omitted[21]. The prototype performs key generation in 12 rounds and conducts one signing attempt in 17 rounds. Here, the *round* is a sequence of actions: (1) the party or parties compute some values locally, (2) either the server, the client, or both concurrently send each other some data, (3) the parties wait until they receive the data that was sent. The Δ values are communicated to the CRP on each connection establishment. A certain limitation regarding the processing of the CRP-generated values is discussed in Section 6.4.

[17] In the earlier stages of development, Dilithium 3.1 was supported, instead of ML-DSA.

[18] `https://github.com/GiacomoPope/dilithium-py`

[19] `https://openquantumsafe.org/liboqs/`

[20] `https://botan.randombit.net/`

[21] For instance, the prototype does not compute characteristic vectors for all the parts in the symmetric reshare to parts protocol, since only some characteristic vectors are needed.

6.2 Used technologies

The prototype is implemented in Python and was tested to work with Python versions 3.10 and 3.12. This programming language was selected for the ease of prototyping.

The prototype heavily relies on the NumPy library to significantly speed up computations involving large arrays. The prototype was tested with NumPy versions 1.26.0 and 2.1.0. Although using NumPy allowed for faster computations, comparatively low performance remained one of the math drawbacks of using Python as the programming language.

The cryptography library `Botan3` is used as a test dependency for verifying the correctness (conformity) of the produced signatures. The Python `galois`[22] library is used in a test of the NTT-based polynomial multiplication.

6.3 Implementation details

The cryptographic hash function used for hashing MAC values (see Protocol 1) and for the hash-based commitment scheme (see Protocols 12 and 16) is SHAKE-256.

Generation of an array from a seed (see Protocols 2 and 16) is performed using a pseudo-random number generator that leverages the same hash function. For array generation, the seed is first absorbed into SHAKE-256. Then, as long as more elements need to be generated, either 1, 2, or 4 bytes (depending on the modulo q of the elements of the array) are squeezed from SHAKE-256. Only l least significant bits of the squeezed value are kept (where $2^{l-1} \leq q \leq 2^l$). The value v is used as a generated value only if $v < q$.

In the prototype, the Δ values are generated indirectly. Each party initially chooses a 256-bit seed d_i uniformly at random. These seeds are then sent to the CRP. Afterward, seed d_i can be deterministically expanded into all the values Δ_i.

Runnable scripts for the Server component and the Client component each instantiate an object of the `MPC` class, which implements the protocols executed by the parties. Hence, most logic is shared between the Server and the Client.

The tests created for the prototype can be categorized into three groups.

- Primitive tests for serialization and deserialization routines, some of the CR generation functions, NTT, NTT^{-1}, the local function `split` and matrix multiplication.
- The MPC protocols test. The test runs the protocols with various input values and verifies that the outputs of the protocols are correct. It covers the Protocols 3–8 and 13–15. To run the protocols, the test instantiates two objects of the `TestMPC` class (which inherits the `MPC` class) — one for each party — and then runs the same methods of these objects in parallel. The `TestMPC` class is designed such

[22] `https://github.com/mhostetter/galois`

that the parties communicate by placing and retrieving bytes from stacks. The Server generates the CR and places the Client's shares in the corresponding stack. Additionally, this class implements traffic logging (refer to Section 6.5), which also supports the key generation and signing iteration protocols.

- Key and signature tests. These tests operate on the objects, saved to the computer storage. The key test verifies consistency between the public key and the two shards of the private key. It also verifies the distribution of values in the private key. The signature tests ensure that the signature matches the public key and the signed message.

6.4 Limitations

For communication, the prototype utilizes "raw" TCP sockets. Thus, the communication between the participants is not authenticated, not integrity protected and not encrypted. Of course, this means that this prototype must not be used to securely create signatures, unless the participants are connected using protected communication channels. An example of such channel would be a WireGuard connection, established between the three participants as one-to-one links. [23]

The prototype is not side-channel resistant.

Another limitation of the prototype is the lack of support for the "eager" streaming as discussed in Section 2.7. In the prototype, the correlated randomness is requested by the Server every time new correlated randomness is needed, thus the client needs to wait for the seed every time a new object is required. Implementing the eager streaming would significantly reduce latency, especially in networks with large packet transmission delay (see Section 6.6).

Furthermore, ML-DSA signing iterations in the prototype are performed consecutively. Performing the iterations concurrently would significantly reduce the expected (average) number of rounds needed for creating a signature.

6.5 Traffic measurements

This section presents the results of the traffic logging of the prototype for the key generation and signing protocols. Note that the traffic volumes in directions S→CRP, C→CRP and CRP→C are insignificant (less than 3kB each for key generation or signing attempt for any parameter set). Due to the prototype implementation details, the traffic volume in directions S→C and C→S is always equal; henceforth the traffic volume

[23] For a potential production system, TLS 1.3 with certificate pinning would be desirable for the CRP-Client connection and the Server-Client connection. Due to the large traffic volume, a dedicated physical one-to-one cable is desirable for the CRP-Server connection. The channel should be protected from sniffing attacks, perhaps by employing encryption with a symmetric cipher.

in either of these two directions is denoted as C↔S (i.e., the total number of bytes exchanged between the Server and the Client is twice this value).

The *actual* values constitute the cumulative volume of data transmitted from one party to another. These values were measured by the MPC protocol test code[24] (see Section 6.3) — note that these values do not reflect the TCP overhead. The *idealized* values represent the information-theoretical limit of compression for the transmitted data. To derive these values, the information transmitted was imagined to consist solely of the array elements, where each element was considered to require $\frac{\log_2(q)}{8}$ bytes to represent, where q is the modulus of the array. In the presented tables, the idealized traffic volume values are displayed in brackets, next to the actual traffic volumes (all values are rounded).

Table 2 shows the traffic measurements for the key generation protocol. Table 3 shows the traffic measurements for one signing attempt (Protocol 18). During the signing process, the traffic outside of signing iterations (i.e., transmitting μ and revealing z to the Client) has a volume not exceeding 10kB for any parameter set.

Table 2. Traffic measurements for the key generation protocol.

	ML-DSA-44		**ML-DSA-65**		**ML-DSA-87**	
Total						
C↔S	1613kB	(962kB)	2784kB	(1048kB)	3024kB	(1803kB)
CRP→S	23.8MB	(16.7MB)	23.0MB	(14.6MB)	44.7MB	(31.4MB)

The obtained measurements demonstrate that the potential commercial roll-out of Duolithium, in which clients would be edge devices (smartphones), would *not* be impractical due to prohibitively high server-client traffic requirements. However, significant throughput requirements are imposed on the CRP-to-server communication channel due to the high volume of traffic.

Full traffic listings (for ML-DSA-44) are provided in the Appendix A.

6.6 Runtime performance

The computational performance of the prototype was tested using Lenovo Thinkpad T490s computers, equipped with Intel Core i5-8265U processors, running Ubuntu 24.04.1 LTS, with Power Mode set to Performance.

[24] As an additional precaution against the possibility of the measuring code being incorrect, the traffic volume for the ML-DSA-44 key generation and the total traffic volume for one signing iteration of ML-DSA-44 were also measured using Wireshark (`https://www.wireshark.org/`). The measured values were consistent with the values obtained from the MPC protocols test code.

Table 3. Traffic measurements for one signing attempt (signing iteration) protocol.

	ML-DSA-44		**ML-DSA-65**		**ML-DSA-87**	
Generating masking vector						
C↔S	2kB	(2kB)	3kB	(3kB)	5kB	(5kB)
CRP→S	1.0MB	(0.7MB)	1.3MB	(1.0MB)	1.9MB	(1.3MB)
Obtaining high bits						
C↔S	18kB	(13kB)	27kB	(19kB)	36kB	(26kB)
CRP→S	10.2MB	(7.4MB)	13.0MB	(9.3MB)	17.3MB	(12.4MB)
Rejection sampling						
C↔S	38kB	(24kB)	51kB	(34kB)	69kB	(44kB)
CRP→S	23.6MB	(17.7MB)	32.4MB	(24.5MB)	44.2MB	(33.4MB)
Total (one attempt)						
C↔S	58kB	(39kB)	81kB	(56kB)	110kB	(75kB)
CRP→S	34.8MB	(25.8MB)	46.7MB	(34.8MB)	63.4MB	(47.1MB)

In total, three test configurations were employed.

- In the first test configuration (`local`), CRP, Server and Client components all ran on a single computer. Communication was performed via localhost IP networking.
- In the second test configuration (`Ethernet`), CRP, Server and Client components ran on three different computers, connected to a local network with Ethernet cables; between each pair of computers, the roundtrip delay (as measured by the `ping` utility) did not exceed 1 ms.
- In the third test configuration (`Wi-Fi`), the setup was deliberately constructed to increase the average latency to the client. This configuration matched the second, except that the client was connected via a Wi-Fi network, shared with a phone, which was in turn connected to the Internet via a the cellular LTE network. The roundtrip delay between the client and the server was measured continuously during the tests: the median roundtrip delay was 35 ms.

For each test configuration, a total of 50 key generation processes and 50 signing processes were performed for each parameter set. All the time intervals were measured by the client. The value in the brackets shows the standard deviation, the value to the left of the brackets shows the average.

Table 4 shows the average measured wall clock time of the key generation process, in seconds.

Table 4. Average run times for the entire key generation process.

	ML-DSA-44		**ML-DSA-65**		**ML-DSA-87**	
Total						
`local`	2.480s	(0.464s)	2.440s	(0.304s)	4.417s	(0.424s)
`Ethernet`	2.652s	(0.043s)	2.837s	(0.051s)	4.963s	(0.073s)
`Wi-Fi`	5.491s	(1.224s)	7.564s	(3.969s)	9.298s	(2.230s)

Table 5 shows the average measured wall clock time (in seconds) for the different parts of the signing attempt, average time for the entire signing attempt.

Table 5. Average run times for the signing iteration.

	ML-DSA-44		**ML-DSA-65**		**ML-DSA-87**	
Generating masking vector						
`local`	0.101s	(0.054s)	0.121s	(0.062s)	0.184s	(0.106s)
`Ethernet`	0.089s	(0.015s)	0.113s	(0.016s)	0.151s	(0.029s)
`Wi-Fi`	0.132s	(0.028s)	0.155s	(0.030s)	0.190s	(0.037s)
Obtaining high bits						
`local`	0.380s	(0.103s)	0.490s	(0.113s)	0.661s	(0.149s)
`Ethernet`	0.438s	(0.015s)	0.551s	(0.016s)	0.712s	(0.024s)
`Wi-Fi`	0.932s	(0.135s)	1.033s	(0.121s)	1.212s	(0.120s)
Obtaining signature candidate						
`local`	0.071s	(0.026s)	0.086s	(0.025s)	0.122s	(0.036s)
`Ethernet`	0.054s	(0.002s)	0.067s	(0.002s)	0.093s	(0.003s)
`Wi-Fi`	0.056s	(0.007s)	0.069s	(0.003s)	0.095s	(0.004s)
Rejection sampling						
`local`	1.822s	(0.333s)	2.562s	(0.411s)	3.483s	(0.475s)
`Ethernet`	1.966s	(0.053s)	2.714s	(0.083s)	3.688s	(0.108s)
`Wi-Fi`	2.360s	(0.107s)	3.155s	(0.100s)	4.241s	(0.128s)
Total (one attempt)						
`local`	2.386s	(0.460s)	3.277s	(0.531s)	4.464s	(0.615s)
`Ethernet`	2.548s	(0.064s)	3.449s	(0.097s)	4.646s	(0.121s)
`Wi-Fi`	3.484s	(0.238s)	4.413s	(0.214s)	5.741s	(0.250s)

Average total time, as well as total median time (in seconds), for obtaining one signature for different scenarios is presented in Table 6. The large standard deviation can be explained by the fact that the number of iterations necessary to obtain one signature varies between runs.

Table 6. Average and median run times for the entire signing process.

	ML-DSA-44		**ML-DSA-65**		**ML-DSA-87**	
Total (one signature), average						
`local`	9.245s	(9.117s)	16.354s	(12.366s)	17.274s	(10.288s)
`Ethernet`	9.144s	(7.725s)	14.990s	(12.886s)	18.140s	(14.154s)
`Wi-Fi`	16.502s	(13.042s)	23.969s	(17.027s)	24.734s	(18.353s)
Total (one signature), median						
`local`	6.848s		14.630s		17.773s	
`Ethernet`	7.716s		10.384s		14.498s	
`Wi-Fi`	12.447s		18.722s		20.701s	

The presented values demonstrate that it is plausible that an optimized implementation of the Duolithium client component capable of producing signatures in near-real time could be created for edge devices (smartphones). [25]

[25] After the Python prototype discussed in this thesis was created, a different, more optimized implementation of Duolithium was developed by a third party, in Rust. Preliminary performance testing concluded that the current at the time of testing version of the prototype is capable of performing one ML-DSA-44 signing iteration on a local machine in half a second.

7 Conclusion

In this work, a novel two-party computation cryptographic scheme capable of producing ML-DSA-compatible public keys and signatures was documented. Some of the sub-protocols that the scheme relies upon were invented as part of the research for this thesis. Functionality proofs are presented for the invented protocols. A prototype that fully implements the scheme was developed and successfully tested for functionality. The practicality of the scheme was demonstrated by measuring the traffic requirements and the computational performance of the developed prototype.

Both the key generation protocol and the signature creation protocol of the documented scheme are secure against potentially malicious Server, potentially malicious Client. Additionally, the key generation protocol is CRP-wary. As part of future work, rigorous security proofs will be given for these security properties, most likely in the universal composability framework. The security implications of the selective failure attack on the presented symmetric reshare to parts protocol will be analyzed in more detail. Security improvements against selective failure attacks will be proposed. The permissibility of the declassification of $\boldsymbol{w}^H$ in the signing protocol will be shown formally.

As part of future work, production-grade code that implements this scheme will be developed, potentially allowing for the rollout of a commercial digital identity solution. This solution could enable authentication and digital signing, relying on this scheme. Using a compiled, low-level programming language (or languages) would allow running the CRP and the server components in Hardware Security Modules and the client component in Android- and iOS-run devices.

The work is already underway on both creating the security proofs as well as creating an optimized implementation of the scheme in a low-level programming language.

Acknowledgements. This work has been supported by Estonian Research Council, grant No. PRG1780. I would like to thank Peeter for his unfading support during the entire duration of the second iteration of the 2PC Dilithium project: for always finding time to explain difficult concepts to me, for the insightful discussions, for providing actionable feedback and giving me guidance. During the project, Peeter always was supportive of my initiative and helped me to achieve the goals. I would also like to thank Antonín and Toomas for their suggestions for improving this work.

References

[1] Estonian ICT Cluster. SplitKey: Secure Digital Identity for Mobile Devices. https://e-estoniax.com/solution/splitkey/ (2025-01-02).

[2] Cybernetica. Introduction to SplitKey Foundations, June 2020. https://cyber.ee/uploads/Split_Key_White_Paper_3506f03c34.pdf (2025-01-02).

[3] Michele Mosca and Marco Piani. Quantum Threat Timeline Report 2022. Technical report, Global Risk Institute, 2022. https://globalriskinstitute.org/publication/2022-quantum-threat-timeline-report/ (2025-01-02).

[4] National Institute of Standards and Technology. Transition to Post-Quantum Cryptography Standards. Technical Report NIST IR 8547 (Initial Public Draft), National Institute of Standards and Technology, Gaithersburg, MD, November 2024. https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf (2025-01-02).

[5] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. Technical Report FIPS 204, National Institute of Standards and Technology, Gaithersburg, MD, August 2024. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf (2025-01-02).

[6] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. https://eprint.iacr.org/2017/633 (2025-01-02).

[7] National Institute of Standards and Technology. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf (2025-01-02).

[8] Peeter Laud. Dilithium-in-2PC. Unpublished, 2024.

[9] Joseph A. Gallian. *Contemporary Abstract Algebra*. Taylor & Francis Group, 10th edition, 2021.

[10] Ardianto Satriawan, Rella Mareta, and Hanho Lee. A Complete Beginner Guide to the Number Theoretic Transform (NTT). Cryptology ePrint Archive, Paper 2024/585, 2024. https://eprint.iacr.org/2024/585.pdf (2025-01-02).

[11] Shi Bai, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1), 2021. https://pq-crystals.org/ (2025-01-02).

[12] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011. https://doi.org/10.1007/978-3-642-20465-4_11 (2025-01-02).

[13] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, volume 576, pages 420–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992. https://doi.org/10.1007/3-540-46766-1_34 (2025-01-02).

[14] Nuttapong Attrapadung, Hiraku Morita, Kazuma Ohara, Jacob C. N. Schuldt, and Kazunari Tozawa. Memory and Round-Efficient MPC Primitives in the Pre-Processing Model from Unit Vectorization. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 858–872, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3488932.3517407 (2025-01-02).

[15] Yehuda Lindell and Benny Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *Advances in Cryptology - EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007. https://doi.org/10.1007/978-3-540-72540-4_4 (2025-01-02).

Appendices

A. Prototype traffic extracts

This appendix contains the traffic listings for ML-DSA-44, obtained by the `TestMPC` code. In both listings, the first column denotes the direction of the traffic, the values in the next two columns denote the actual traffic volume and the rounded idealized traffic volume for the sent data object (in bytes). The final column contains the description of the sent object. For CRP traffic, it is the CR description, which consists of the requested CR dimensions, CR type, and requested MACs (`lf` for both MACs, `l` for only Server MACs). For S↔C traffic, it is a description of the point in the protocol at which the declassification is performed.

Key generation (Protocol 16).

```
CRP->S |    344213 |     99862 | (56, 8, 1, 256, 1) x mult_5 ()
CRP->C |       308 |        96 | (56, 8, 1, 256, 1) x mult_5 ()
  S->C |    229527 |     66639 | generate_s12>entry>leader_mac>mult>alpha_beta
  C->S |    229527 |     66639 | generate_s12>entry>leader_mac>mult>alpha_beta
CRP->S |    344213 |     99862 | (56, 8, 1, 256, 1) x mult_5 ()
CRP->C |       308 |        96 | (56, 8, 1, 256, 1) x mult_5 ()
  S->C |    229527 |     66639 | generate_s12>entry>follower_mac>mult>alpha_beta
  C->S |    229527 |     66639 | generate_s12>entry>follower_mac>mult>alpha_beta
CRP->S |    764014 |    449866 | (8, 1, 256, 1) x cv_5_8380417 (lf)
CRP->C |       212 |        64 | (8, 1, 256, 1) x cv_5_8380417 (lf)
  S->C |        86 |        32 | generate_s12>cv_zc>seed
  C->S |        86 |        32 | generate_s12>cv_zc>seed
  S->C |       316 |       216 | generate_s12>m_seed>shared_bytes (&commitment)
  C->S |       316 |       216 | generate_s12>m_seed>shared_bytes (&commitment)
CRP->S |  22364327 |  16073255 | (8, 1, 256, 1, 10, 7) x mult_8380417 (lf)
CRP->C |       338 |        96 | (8, 1, 256, 1, 10, 7) x mult_8380417 (lf)
  S->C |   1147039 |    824334 | generate_s12>mult>alpha_beta
  C->S |   1147039 |    824334 | generate_s12>mult>alpha_beta
  S->C |       102 |        32 | generate_s12>mult_zc>seed
  C->S |       102 |        32 | generate_s12>mult_zc>seed
  S->C |       215 |       133 | generate_s12>zc_open
  C->S |       215 |       133 | generate_s12>zc_open
  S->C |      2120 |       627 | generate_s12>masked_keyshare
  C->S |      2120 |       627 | generate_s12>masked_keyshare
  S->C |       188 |       124 | rho>shared_bytes (&commitment)
  C->S |       188 |       124 | rho>shared_bytes (&commitment)
  S->C |      4164 |      2976 | t_declassify
  C->S |      4164 |      2976 | t_declassify
```

Signing attempt (Protocol 18).

```
CRP->S |    958517 |    688854 | (18432,) x bits_8380417 (lf)
CRP->C |        90 |        32 | (18432,) x bits_8380417 (lf)
  S->C |      2364 |      2336 | generate_y>flipped_bits
  C->S |      2364 |      2336 | generate_y>flipped_bits
CRP->S |   9555534 |   6865317 | (4, 1, 256) x partsaa_31;24;16;16;89 (lf)
CRP->C |      1256 |       448 | (4, 1, 256) x partsaa_31;24;16;16;89 (lf)
```

```
  S->C |      1092 |       795 | highbits>parts>b0
  C->S |      1092 |       795 | highbits>parts>b0
  S->C |      1092 |       747 | highbits>parts>b1
  C->S |      1092 |       747 | highbits>parts>b1
  S->C |      1092 |       672 | highbits>parts>b2
  C->S |      1092 |       672 | highbits>parts>b2
  S->C |      1092 |       672 | highbits>parts>b3
  C->S |      1092 |       672 | highbits>parts>b3
  S->C |      1092 |       989 | highbits>parts>b4
  C->S |      1092 |       989 | highbits>parts>b4
  S->C |        78 |        32 | highbits>parts>security_check>seed
  C->S |        78 |        32 | highbits>parts>security_check>seed
  S->C |       108 |        67 | highbits>parts>security_check_declassify
  C->S |       108 |        67 | highbits>parts>security_check_declassify
CRP->S |    426096 |    306158 | (4, 1, 256) x eqzero_8380417_6_8380417 (lf)
CRP->C |       206 |        64 | (4, 1, 256) x eqzero_8380417_6_8380417 (lf)
  S->C |      4164 |      2976 | highbits>correction>d
  C->S |      4164 |      2976 | highbits>correction>d
CRP->S |    266352 |    191349 | (4, 1, 256) x eqzero_8380417_3_8380417 (lf)
CRP->C |       206 |        64 | (4, 1, 256) x eqzero_8380417_3_8380417 (lf)
  S->C |      4164 |      2976 | highbits>security_check_eqzero>d
  C->S |      4164 |      2976 | highbits>security_check_eqzero>d
  S->C |        80 |        44 | highbits>security_check_declassify
  C->S |        80 |        44 | highbits>security_check_declassify
  S->C |      4164 |      2976 | wh_declassify
  C->S |      4164 |      2976 | wh_declassify
CRP->S |    190573 |    118399 | (4, 1, 256) x partsap_15_6_29 (l)
CRP->C |       200 |        64 | (4, 1, 256) x partsap_15_6_29 (l)
CRP->S |    190573 |    118399 | (4, 1, 256) x partsap_15_6_29 (l)
CRP->C |       200 |        64 | (4, 1, 256) x partsap_15_6_29 (l)
  S->C |      8327 |      5952 | rejection>parts>leftover_forward
  C->S |      8327 |      5952 | rejection>parts>leftover_forward
CRP->S |  17080426 |  12847427 | (6, 8, 2, 256) x cv_29_67 (l)
CRP->C |       208 |        64 | (6, 8, 2, 256) x cv_29_67 (l)
  S->C |     24648 |     14956 | rejection>comp>s_ovf>d
  C->S |     24648 |     14956 | rejection>comp>s_ovf>d
CRP->S |   6131810 |   4712575 | (8, 2, 256) x cv_67_71 (l)
CRP->C |       192 |        64 | (8, 2, 256) x cv_67_71 (l)
  S->C |      4164 |      3138 | rejection>comp>l_ovf>d
  C->S |      4164 |      3138 | rejection>comp>l_ovf>d
CRP->S |      2105 |      1446 | (1, 1) x cv_71_8380417 (l)
CRP->C |       181 |        64 | (1, 1) x cv_71_8380417 (l)
  S->C |        65 |        33 | rejection>cv>d
  C->S |        65 |        33 | rejection>cv>d
  S->C |        68 |        35 | rejection>open
  C->S |        68 |        35 | rejection>open
```

B. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Semjon Kravtšenko**,
*(*author's name)

1. grant the University of Tartu a free permit (non-exclusive licence) to:

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis

   Efficient Two-Party ML-DSA Protocol in Active Security Model,
   *(*title of thesis)

   supervised by Peeter Laud and Toomas Krips.
   *(*supervisors' names)

2. I grant the University of Tartu the permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work from **2025-01-23** until the expiry of the term of copyright,

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Semjon Kravtšenko
2025-01-09

Parts of this thesis describe an invention, which is protected by a patent or is in the process of being patented.