Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Constant-Time Implementation of NTS-KEM

Master's Thesis in Computer Science by

Jan Gilcher

March 23, 2020

Supervisor: Professor Kenny Paterson

Abstract

In this thesis we present a constant-time implementation of NTS-KEM, a post-quantum secure KEM based on the McEliece and Niederreiter PKE schemes and currently taking part in the NIST post-quantum cryptography standardization process. We verified our implementation using ctgrind and show that we achieve performance suitable for practical use. We present several techniques to achieve security against timing attacks, and provide insights into the implementation of fast constant-time algorithms.

Contents

1. Introduction

The advent of quantum computing will usher in successful attacks on till then secure cryptographic schemes, due to the different computational model of quantum computers. Fortunately, cryptographers have been researching the impact of this on current cryptographic schemes as well as how to build new, secure cryptographic schemes with this threat in mind. This has resulted in several new cryptographic schemes.

However, implementations of cryptographic schemes have a long history of being insecure even if the underlying cryptographic protocols are secure. These issues range from normal bugs to abstractions in the theoretical model that are not valid in the real world. One of these issues are so called timing side-channels.

The US National Institute of Standards and Technology (NIST) has also been aware of the threat posed by quantum computers and started a standardization process in 2017 [38]. This process has reached its second round in 2019 and will soon reach its third round.

1.1. Contribution

In this thesis we focused on NTS-KEM [3], a post-quantum secure key-encapsulation mechanism by Albrecht et al. that is one of the proposals in the second round of the NIST post-quantum cryptography standardization process. While the NTS-KEM team took care to provide a secure implementation, the reference implementation is not fully secure against timing attacks. Our goal was to implement a version of NTS-KEM that is secure against timing attacks, and if possible to achieve this without changes to the NTS-KEM specification. Furthermore we also aim for high performance, to ensure that NTS-KEM is competitive and usable in practice in a secure way.

1.2. Outline

This thesis is structured as follows. We start with a short note on notation and prerequisites below. We will then briefly introduce some cryptographic definitions and provide some background on side-channels in Chapter 2. Section 2.3 then gives an introduction into the techniques used to prevent timing based side-channels. We follow this up in Chapter 3 with a discussion of NTS-KEM and its implementation as well as timing issues present in it. In Chapter 4 we will present

our implementation, followed by a performance evaluation in Chapter 5. In Appendices A and B we provide some supplementary proofs. Furthermore during the work on this thesis we identified several bugs in the NTS-KEM [3] implementation which are listed in Appendix C.

1.3. Notation and Prerequisites

In the following chapters we assume that the reader is familiar with C, as C will be used for code snippets. We also assume that the reader is familiar with integer representations in modern computers.

All logarithms are logarithms to the base 2 if not explicitly stated otherwise, i.e. we write log instead of $\log_2$. $\mathbb{N}_m$ denotes the subset of $\mathbb{N}$ with elements smaller than m. We include 0 in $\mathbb{N}$. We denote the field with two elements as $\mathbb{F}_2$, and an extension field of $\mathbb{F}_2$ with 2^m elements as $\mathbb{F}_{2^m}$. For any field $\mathbb{F}$, $\mathbb{F}[x]$ is the ring of univariate polynomials over $\mathbb{F}$. We use common notation for vectorspaces, spaces of matrices and matrix transposition (i.e. $\mathbb{F}_2^n$, $\mathbb{F}_2^{k\times n}$, and $\mathbf{G}^\mathbf{T}$ respectively). Vectors are typeset using bold lowercase ($\mathbf{e}$) and bold uppercase is used for matrices ($\mathbf{G}$). The i-th row of matrix $\mathbf{G}$ is denoted by $\mathbf{G}_i = (g_{i,0}, g_{i,1}, \ldots, g_{i,n-1})$. We use $\mathrm{hw}(\mathbf{e})$ to denote the *Hamming weight* of a vector $\mathbf{e}$. Concatenation of vectors is denoted by $(\cdot \mid \cdot)$, i.e $(\mathbf{v} \mid \mathbf{w}) \coloneqq (v_0, ..., v_{n_1-1}, w_0, ..., w_{n_2-1})$.

2. Background

While we assume some familiarity with basic cryptographic concepts, we will give the relevant cryptographic definitions in Section 2.1. Furthermore this chapter introduces the basics of side-channels (Section 2.2), focusing on timing attacks and cache-timing attacks. We will then introduce concepts of constant-time programming as a way to prevent such attacks in Section 2.3.

2.1. Cryptography

In cryptography we distinguish between symmetric and asymmetric cryptography. Symmetric cryptography uses one key that is used both for encrypting and decrypting a message. We omit a detailed discussion of symmetric encryption as it is not relevant to the issues discussed in this thesis.[1]

In contrast to symmetric cryptography, asymmetric (or public-key) cryptography uses two distinct keys, a public key used for encryption and a private (or secret) key used for decryption.

Definition 2.1.1 (Public-key Encryption Scheme)**.** A *public-key encryption (PKE) scheme* consist of:

- a randomized key-generation function, $\mathsf{KGen} : \emptyset \to \mathcal{P} \times \mathcal{S}$,
- an encryption function, $\mathsf{Enc} : \mathcal{M} \times \mathcal{P} \times \mathcal{R} \to \mathcal{C}$,
- a decryption function, $\mathsf{Dec} : \mathcal{C} \times \mathcal{S} \to \mathcal{M} \cup \{\perp\}$,[2]

so that the following holds:

$$\forall m \in \mathcal{M},\ r \in \mathcal{R},\ (\mathsf{pk}, \mathsf{sk}) = \mathsf{KGen}() : \mathsf{Dec}(\mathsf{Enc}(m, \mathsf{pk}; r), \mathsf{sk}) = m$$

We call $\mathcal{P}$ the public-key space, $\mathcal{S}$ the private-key space,[3] $\mathcal{M}$ the message space, $\mathcal{C}$ the ciphertext space and $\mathcal{R}$ the randomness space.

As the name implies, the public key can be released to the public while the private key is kept secret. This allows anybody to send encrypted messages to the entities in possession of the corresponding private key.

[1] Interested readers not familiar with these concepts can find formal definitions in common introductory literature like *Introduction to Modern Cryptography* by Katz and Lindell [26].
[2] $\perp$ denotes a decryption error.
[3] Also called secret-key space.

To enable security proofs, a PKE scheme is often parameterized with a security parameter n, usually relating to the length of the (private) key. The usual security definition requires that an adversary is not able to decide whether a ciphertext c is the encryption of a message m_0 or m_1, both chosen by the adversary, even when given access to a decryption oracle. Formally this is captured in the following definitions.

Definition 2.1.2 (IND-CCA for PKE)**.** Consider the following game:[4]

1. The challenger generates a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ and the public-key pk is given to the adversary.
2. The adversary can obtain the decryption of any ciphertexts of his choosing.
3. The adversary chooses two messages m_0 and m_1 of the same length.
4. The challenger chooses a random bit $b \in \{0, 1\}$ and sends $c = \mathsf{Enc}(m_b, \mathsf{pk})$ to the adversary.
5. The adversary can repeat step 2. They may not ask for decryption of c.
6. The adversary outputs a bit $b' \in \{0, 1\}$.

Then a PKE scheme Π with security parameter n is called IND-CCA[5] secure, if for any efficient adversary

$$f(n) = \Pr_{PKE(\Pi,n)}[b = b'] - \frac{1}{2}$$

is a negligible function.[6] Here $\Pr_{PKE(\Pi,n)}[\cdot]$ denotes the probability space defined by the above game, when instantiated with the scheme Π and the security parameter n.

Definition 2.1.3 (IND-CPA for PKE)**.** A PKE scheme Π is called IND-CPA[7] secure, if the conditions of Definition 2.1.2 hold, when steps 2 and 5 are removed from the game.

Note that IND-CCA security implies IND-CPA security.

Public-key encryption is usually based on number theory or other algebraic structures and the security proofs rely on assumptions about the hardness of underlying mathematical problems, e.g. factoring integers or computing the discrete

[4] A game in a security definition is played by two distinct entities, an adversary trying to break the scheme and an honest challenger.
[5] Indistinguishable under Chosen-Ciphertext Attack.
[6] A function f is negligible, if it grows slower than any inverse polynomial or formally $\forall c \in \mathbb{N}\ \exists n\ \forall x > n : |f(x)| < \frac{1}{x^c}$.
[7] Indistinguishable under Chosen-Plaintext Attack.

logarithm. For example, RSA [42] is easy to break if factoring integers into prime numbers is computationally easy.

Due to being based on algebraic structures PKE schemes are often significantly more computationally expensive than symmetric cryptography based on block ciphers. On the other hand public-key cryptography makes key distribution easier, since the encryption key does not have to be kept a secret and can be openly shared. For this reason encryption is often done using symmetric encryption schemes, where public-key cryptography is used to ensure that all parties have a secret symmetric key, for example by using a key-agreement protocol or a key encapsulation mechanisms.

Definition 2.1.4 (Key Encapsulation Mechanism)**.** A *key encapsulation mechanism (KEM)* consists of:

- a randomized key-generation function, $\mathsf{KGen} : \emptyset \to \mathcal{P} \times \mathcal{S}$,
- an encapsulation function, $\mathsf{Enc} : \mathcal{P} \times \mathcal{R} \to \mathcal{K} \times \mathcal{C}$,
- a decapsulation function, $\mathsf{Dec} : \mathcal{C} \times \mathcal{S} \to \mathcal{K} \cup \{\perp\}$,

so that the following holds:

$$\forall r \in \mathcal{R},\ (\mathsf{pk}, \mathsf{sk}) = \mathsf{KGen}() : (c, \mathsf{k}) = \mathsf{Enc}(\mathsf{pk}; r) \longrightarrow \mathsf{Dec}(c, \mathsf{sk}) = \mathsf{k}$$

We call $\mathcal{P}$ the public-key space, $\mathcal{S}$ the private-key space, $\mathcal{K}$ the symmetric-keyspace,[8] $\mathcal{C}$ the ciphertext space, and $\mathcal{R}$ the randomness space.

Note that a key encapsulation mechanism is not a public-key encryption scheme. It does not grant the ability to encrypt arbitrary messages directly. However one can construct a PKE scheme from a KEM by combining it with a symmetric encryption scheme, that makes use of the encapsulated key, often called KEM-DEM.[9] It is also possible to construct a KEM from a PKE scheme, when the PKE scheme's message space is large enough.

For a KEM, security is defined by the adversary's inability to decide whether a given ciphertext is the encapsulation of a given key. This is formally stated by the following definitions:

Definition 2.1.5 (IND-CCA for KEM)**.** Consider the following game:

1. The challenger generates a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ and sends pk to the adversary.
2. The challenger generates a random $r \in \mathcal{R}$ and $(\mathsf{k}_1, c) = \mathsf{Enc}(\mathsf{pk}; r)$, and picks a random k_2 with the same length as k_1, and a random bit $b \in \{0, 1\}$. They then send (k_b, c) to the adversary.

[8] also encapsulated-key space.
[9] from DEM – Data Encapsulation Mechanism.

3. The adversary can obtain the decapsulation of any ciphertexts of his choosing, except c.

4. The adversary outputs a bit $b' \in \{0, 1\}$.

Then a KEM Π with security parameter n is called IND-CCA secure, if for any efficient adversary

$$f(n) = \Pr_{KEM(n)}[b = b'] - \frac{1}{2}$$

is a negligible function. $\Pr_{KEM(n)}[\cdot]$ denotes the probability space defined by the above game, when instantiated with the scheme Π and the security parameter n.

Definition 2.1.6 (IND-CPA for KEM)**.** A KEM is called IND-CPA secure if the conditions of Definition 2.1.5 hold when step 3 is removed from the game.

Post-Quantum Cryptography

While the mathematical structure of public-key cryptography is highly beneficial for provably secure cryptography, the rise of quantum computers threatens their security.

The security of many common public-key cryptographic algorithms relies on the hardness assumptions about certain computational problems, often in the form of a function f, where $f(x; \mathsf{pk})$ is easy to compute but $f^{-1}(y; \mathsf{sk})$ is difficult to compute without knowledge of sk. (Note that such assumptions are in NP.)

Since it is not known whether BQP, the set of all problems solvable by a quantum computer in polynomial time with bounded error, is equal to NP, not all public key cryptography is automatically broken. In fact it is believed that there are problems that are suitable for cryptographic hardness assumptions.

However, a working quantum computer breaks some common hardness assumptions due to Shor's algorithm [44], which can be used to factor a large number into primes, or compute the discrete logarithm.

2.2. Side-Channel Attacks

Side-channel attacks usually arise when physical properties of the execution, which are absent from the theoretical model, can leak secret information to the attacker. Many physical properties can be used to construct side-channels, like timing [29], power consumption [30], sound [19], heat [23], and electromagnetic emissions [36].

While most of these require the attacker to be close to their victim, timing based side-channels can in principle be observed and exploited remotely over the Internet. Therefore this thesis only treats timing based side-channels.

Listing 2.1 – Vulnerable array comparison.

```
1  int arrayEquals(char* a, char* b, int len)
2  {
3    for (int i = 0; i < len; i++) {
4      if (a[i] != b[i])
5        return 0;
6    }
7    return 1;
8  }
```

Timing Attacks

Timing attacks were the first published side-channel attack, namely an attack leveraging a timing difference due to the square-and-multiply method used in the modular exponentiation in Diffie-Hellman, RSA and other systems at the time, published in 1996 by Kocher [29]. A simple example of a timing attack is array equality checking (see Listing 2.1). In this example we restrict ourselves to the case of two arrays of the same length. It is easy to see that the runtime of the function `arrayEquals` is directly dependent on how many elements have to be compared until a difference in the strings is found. Put another way, the runtime of the function depends only on the length the longest common prefix of `a` and `b`.

For an attack sketch, assume the secret `b = "password"`. Then, by trying strings of the form `"aaaaaaaa"`, `"baaaaaaa"`, ... the attacker can measure the timing difference between `a = "aaaaaaaa"` and `a = "paaaaaaa"` and conclude that the first character is `p`. Thus they can infer the whole password by inferring the characters individually, reducing the complexity from 256^n to $256 \cdot n$.[10] Even such a simple example as non-constant-time array comparison is practically relevant, e.g. until 2009 Java's standard library had a timing vulnerability in its method `java.security.MessageDigest.isEqual`, which is used to compare hashes [25].

Cache-Timing Attacks

Cache-timing attacks leverage timing differences induced by the cache hierarchy of modern computers, to infer secrets from secret dependent memory accesses. They were first described in 2002 by Page [39]

Caches store recently accessed data in a smaller, faster memory unit to speed up data access times, which are usually orders of magnitudes slower than other CPU instructions. To increase performance further modern computers usually employ

[10] This example overly simplifies the reality of password cracking. In practice, passwords are usually not chosen uniformly at random over the space of all possible values of the underlying data type. Thus the base might be significantly smaller than 256. This example can be viewed as the more general problem of comparing byte arrays.

a hierarchy of caches with increasing size and access time, where each layer in this hierarchy caches accesses to the next higher level. Essentially caches are a table consisting of fixed size entries to store the data, where a line in the table is called a cache line. The data in the cache is identified using the cache line and a cache tag, both of which are computed from the address of the accessed memory. This can result in different elements of a data structure, like an array, to live in different levels of the hierarchy, and thus access times are not uniform, but instead dependent on the address.

An example relevant to cryptography is the use of look-up tables, as found in fast implementations of AES [17]. AES uses a 256-byte table (S-boxes) to substitute the input bytes with bytes from the tables, where the substitution pattern is defined by the secret key. Thus the access pattern depends on the key. By measuring timing differences between different input messages (and additional knowledge from precomputation on similar hardware) an attacker can then infer the secret index used to access the table and hence information about the key. In 2004 Bernstein [9] executed such an attack against the OpenSSL AES implementation.

2.3. Constant-Time Programming

The term *constant-time programming* is used to describe the collection of programming techniques used to ensure the absence of timing side-channels. In the following we will show techniques to ensure that the control flow of a program is independent from any secrets. We will also discuss how to ensure that basic operations like comparisons are executed in constant-time. Moreover we will discuss additional complications arising from compilers. Finally, we will give a short overview over tools to automatically verify constant-time properties, that might stop the programmer from falling into one of the many pitfalls of constant-time programming.

Branches

One of the main causes of side-channel issues are branches. Listing 2.1 gave an example of this. In that case our branch causes the number of loop iterations to be different depending on the secret. To avoid this our loop variables may not depend on secrets. This is already the case in Listing 2.1, though. The issue is actually an early break (or in this case a return) in the loop body. Thus we also need to omit early breaks, to ensure that loops always have a bound that is either fixed or at least only dependent on a public run-time value (as we can clearly not get rid of all dynamic bounds, e.g. the runtime of any encryption or hash algorithm that is able to work on arbitrarily long inputs will always depend on the length of the input).

Applying these fixes to our string compare results in the program shown in Listing 2.2. Here we use an error flag `err` to keep track of whether the arrays

Listing 2.2 – Second version of array comparison.

```
1  uint32_t arrayEqualsV2(uint8_t* a, uint8_t* b, int32_t len)
2  {
3    uint8_t err = 0;
4    for (int32_t i = 0; i < len; i++) {
5      err |= a[i] != b[i];
6    }
7    return (uint32_t)!err;
8  }
```

Listing 2.3 – Modular exponentiation using *square-and-multiply*.

```
1  // selectBit(x, k) returns the k-th bit of n,
2  // where the most significant bit is the 0-th bit.
3  exp(y, x, n)
4  {
5    s = 1;
6    for (k = 0; k < bitlength(x); k++) {
7      s = (s * s) % n;
8      if (selectBit(x, k)) {
9        s = (s * y) % n;
10     }
11   }
12   return s;
13 }
```

elements differ while we iterate over all elements. At the end we can then simply return the negation of the error flag to indicate whether the arrays are equal.

Not only loops are problematic, though. In the end every `jump`-instruction in the assembly that depends on secrets is dangerous. This is obviously also the case for any `if`/`if-else` constructs, as was the case in Kocher's original attack on RSA [29]. The vulnerable RSA implementation used an approach called *square-and-multiply*[11] to efficiently compute $y^x \mod n$, shown in Listing 2.3. Here Line 9 is only executed when condition in Line 8 is true, i.e. the k-th bit is set, and thus this method directly leaks the hamming-weight of x. While we omit the details of the attack, note that in 1998 Dhem et al. [16] were able to turn this into a practical attack.

Such an attack is fixed by eliminating the branch and ensuring the operations are always executed. The result can then be conditionally written to the variable constant-time conditional assignment. Such an assignment can be realized in several ways, the fastest is using a constant-time conditional move (`cmove`) instruction when the hardware provides such an instruction (like modern x86 processors). A

[11] also *binary exponentiation* or *exponentiation by squaring*.

platform independent way to achieve this is implementing a `mux` utilizing bitwise operations:

```
/**
 * c is either 0 or 1.
 * returns a if c is 1, b otherwise.
**/
uint64_t mux(uint64_t c, uint64_t a, uint64_t b) {
  return b ^ (-c & (a ^ b));
}
```

Here we use the fact that `b ^ b = 0` and thus `a ^ b ^ b = a`, as well as the commutativity of `^`. Additionally computing `-c & a` is conditionally returning `a` if `c` is 1 and 0 if `c` is zero. The minus simply broadcast the 1 in the 0-th bit of `c` to all bits by using that the two's complement representation of -1 is an all-one bitstring. Putting all this together the return expression is either `b ^ a ^ b = a` if `c` is one or `b ^ 0 = b` if `c` is 0.

Bitwise Operations

Hardware instructions provided by the CPU are a source of additional issues when trying to implement constant-time cryptography. Modern CPUs provide a big set of instructions, however, not all of them execute in constant-time. The canonical example of this is integer division (which usually extends to mod, often using the same instruction). While addition and subtraction are easy to implement fast, division is until today still comparatively slow. For this reason many division implementations take shortcuts wherever they can, which results in execution times that are highly dependent on the inputs. This used to be the case for multiplication, too, which was often faster when the upper half of an operand was zero. On modern x86 processors (especially Intel and AMD) this is not the case anymore, and therefore we will not discuss multiplication.

Other issues stem from gaps in the language standards. The C-standard usually does not define any mapping between high-level C-operations and hardware instruction. This leads to the fact that operations might be implemented in non-constant-time. While this is potentially the case for any operation, some operations like comparators or logical boolean operators, are more prone to non-constant-time implementations than others (e.g. addition, subtraction, or bitwise logical operators like `&`, `|`,`~`,`^`) due to a lack of directly corresponding assembly instructions. In some cases seemingly simple operations are even defined inherently with non-constant-time semantics. The most import example of this are the two logical operators `&&` and `||`, whose short-circuit-evaluation semantics[12] necessitate a jump depending on the value of the left operand.

The best way to solve these problems depends on the assumption that the most

[12] The right operand is not evaluated if the result is already clear from evaluating the left operand.

basic operations (addition, subtraction, and bitwise logical operators) will always be translated to directly corresponding hardware instructions. In the following we will present solutions to these problems, these are mostly based on Pornin's *BearSSL* [40] as well as the *Guidelines for low-level cryptography software* by Aumasson [7].

Division is the most difficult one. For divisions by powers of two, a shift can be used, and some divisions can be decomposed into several shifts by powers of two combined with additions and subtractions.[13] This is difficult when the dividend is only known at run time, though. In the case that such a transformation is not possible, algorithms have to be changed to get by without needing secret dependent division. For some well known problems there are solutions, e.g. the multiplication modulo n in modular exponentiation (see Listing 2.3) can be eliminated via a Montgomery multiplication [35].

Logical boolean operators can usually just be directly replaced by the corresponding bitwise operators. Sometimes care has to be taken to ensure that the result is either 0 or 1, which is not guaranteed by the bitwise boolean operators. For this an `isNotZero` check can be implemented in the following way:

```
uint64_t isNotZero(uint64_t a) {
  return (b | -b) >> 63;
}
```

This utilizes the fact that in two's complement representation the most significant bit of an integer stores the sign, and for all integers a other than 0 either $-a < 0$ or $a < 0$ and thus the sign bit is set. Note that the implementation depends on the size of the integer (because of the shift amount) and thus we have to implement different version for all integer sizes we need by changing the shift amount accordingly.

The logical not can easily be implemented using xor:

```
uint64_t not(uint64_t a) {
  return a ^ 1;
}
```

Comparators Checking inequality can be easily implemented using xor similarly to the logical not. Checking for equality can then be implemented based on inequality by logical negation.

```
uint64_t isNotEqual(uint64_t a, uint64_t b) {
  return isNotZero(a ^ b);
}
```

[13] In fact such transformations are regularly done by compiler optimizations, due to the high cost of divisions.

```
uint64_t isEqual(uint64_t a, uint64_t b) {
  return not(isNotEqual(a, b));
}
```

The only comparators missing are the order relations on integers. Strictly speaking we only need to find one (e.g. $<$) and can then express all the remaining ones by combining it with not and comparison with zero. However this results in functions requiring more instructions than what can be achieved if we define them separately. We also have to take care of signed and unsigned integers separately and similarly to `isNotZero` we need different versions for different integer sizes. First we will start with comparison with zero, which is only necessary for signed integers. The easy cases simply involve checking the sign bit:

```
uint64_t isLessThanZero(uint64_t a) {
  return a >> 63;
}

uint64_t isGreaterThanOrEqualZero(uint64_t a) {
  return ~a >> 63;
}
```

To check $b \geq 0$ we cannot simply check the sign bit since zero also has the sign bit set to zero. Instead we can check that $-b \leq 0$. However, negation in two's-complement representation has two fixed-points, one at zero and one at INT-MIN. Thus we have to check that the sign bit is not set and its negation's sign bit is set

```
uint64_t isGreaterThanZero(int64_t a) {
  uint64_t b = a;
  return (~b & -b) >> 63;
}
```

Similarly for $b \leq 0$ we have to check that the sign bit is set, or that it is zero. This we can simplify to check whether $-b$ is non-negative. Note that the second check would again not be sufficient because of INT-MIN being a fix-point of negation:

```
uint64_t isLessThanOrEqualZero(int64_t a) {
  uint64_t b = a;
  return (b | ~-b) >> 63;
}
```

For the generic comparisons $a < b$ it seems tempting to just build them from comparisons with zero, i.e. checking $a - b < 0$. However for integers on a computer those two checks are unfortunately not equivalent due to integer overflow. But we can observe that integer overflow can only happen if they have different signs. If the sign bits are different, it suffices to check the sign bit of a, though, since it is only smaller than b if a is negative and b is non-negative. In the other case we do not have overflow and can simply return the sign bit of $a - b$. Putting that

together we get the following pseudocode:

$$\textsf{if } \operatorname{sign}(a) \neq \operatorname{sign}(b) \textsf{ then return } \operatorname{sign}(a); \textsf{ else return } \operatorname{sign}(a-b); \textsf{ end} \tag{2.1}$$

To implement this in constant-time we can utilize a bitwise version of the mux above. From the less-than relation we can then build the other relations. Note that for the mux we use unsigned integers which results in a type conversion. Remember that C performs conversion from signed to unsigned integers as follows. If x is the value represented by the variable `int x` then the value x' represented by `(unsigned int) x` is computed as $x' = x \mod (MAX\_UNSIGNED\_INT + 1)$. On a two's complement machine and when converting integers of the same size this results in x and x' having the same binary representation in memory.

```
/**
 * if bit i of c is 1, then bit i of return value is equal to
    ↪ bit i of a,
 * otherwise bit i of return value is equal to bit i of b.
**/
uint64_t bitMux(uint64_t c, uint64_t a, uint64_t b) {
  return b ^ (c & (a ^ b));
}

uint64_t isLessThan(int64_t a, int64_t b) {
  uint64_t c = a - b;
  return bitMux(a^b, a, c) >> 63;
}

uint64_t isGreaterThan(int64_t a, int64_t b) {
  return isLessThan(b, a);
}

uint64_t isLessThanOrEqual(int64_t a, int64_t a) {
  return not(isGreaterThan(a, b));
}

uint64_t isGreaterThanOrEqual(int64_t a, int64_t a) {
  return not(isLessThan(a, b));
}
```

For unsigned integers we can in principle utilize the same trick, as overflow occurs in the same instances, i.e. when the most significant bits are different. However for unsigned integers a number with MSB set is bigger than a number with the MSB not set. Thus for $a < b$ we return the sign of b instead:

```
uint64_t isLessThan(uint64_t a, uint64_t b) {
  uint64_t c = a - b;
  return bitMux(a^b, b, c) >> 63;
}
```

Listing 2.4 – Constant-time array comparison.

```
1  uint32_t ct_arrayEquals(uint8_t* a, uint8_t* b, int32_t len) {
2    uint8_t err = 0;
3    for (int32_t i = 0; i < len; i++) {
4      err |= a[i] ^ b[i];
5    }
6    return ~((uint32_t)(err | -err)) >> 31;
7  }
```

The other relations can than be constructed from `isLessThan` in the same way as in the signed case.

With the above knowledge, we can also see that our second approach to make array comparison constant-time (see Listing 2.2) might be non-constant-time since we are using `!=` and `!`. But we also know how to fix this, as shown in Listing 2.4.

Bitslicing

Bitslicing is a programming technique to implement virtual circuits on general purpose computers and is thus closely related to the design of digital circuits. The general idea is to implement a circuit by using elementary bit-wise operations as basic gates. For this the input data is represented in an orthogonalized fashion, i.e. an 8-bit integer would be distributed over 8 n-bit integers, each corresponding to one bit of the input. This was originally used as a performance optimization, since an orthogonal representation of the inputs allows for parallel computation of n instances of the circuit. Usually n is at least the word size of the processor, but can also be higher if the processor provides vector instructions like AVX2.

It turns out however that if one is careful to only use bit-wise operations, this incidentally fulfills the same restrictions as our bit tricks used above to replace branches and potentially variable-time instructions. It thus naturally leads to constant-time programs.

Unfortunately generating circuits for arbitrary problems is hard. Additionally, not all problems are easy to bitslice in a fast way, e.g. many bitsliced versions of AES have struggled to achieve the performance of non-bitsliced AES implementations [11, section 1 – Bitslicing].

Compilers

As explained above high-level programming languages usually do not include runtime in their semantics. Hence compilers provide no guarantees about the runtime of a program. Instead they try, depending on the compiler settings very aggressively, to make programs as fast as possible. The results can differ vastly between compilers. This can be beneficial to performance of cryptographic software, but

its unpredictability is detrimental to security. Take as example the functions in Listing 2.5. Without optimization all functions are compiled as one would expect, with `cond` implementing the branch using jumps. But when enabling optimization both gcc and clang use the `cmov` instruction to implement the branch in `cond`, which makes the function constant-time! However, there is no guarantee that this will be the case for all versions of gcc or clang. Additionally from line 16 clang is able to infer that c is always either zero or one, which then enables it to use a `cmov` instruction as can be seen from Listing 2.6.

Compilers can be even smarter, as can be seen from `ct_cond3`,[14] which uses a different method of expressing the ternary operator using bit operations. Here the knowledge that c is either zero or one enables clang to fully infer that this is simply an implementation of the ternary operator, and thus it compiles to the same instructions as `cond`. While in theses cases the final versions do indeed compile to constant-time code, there is no guarantee for this in general. If the compiler found a more efficient way to implement `cond` using jumps it is reasonable to assume that `ct_cond3` might be implemented in this way, too. Thus we not only need to take care that we do not use any non-constant-time operations. But we also have to take care that our implementation is obfuscated enough so that the compiler cannot fully understand its semantics and thus produces constant time assembly.

With this knowledge, it is also apparent why we must get rid of branches, even though one might think it sufficient to ensure that all branches take the same time. A compiler will not take time differences between branches into consideration and thus will reduce the time needed to take one branch whenever it can.

Verification of Constant-Time Properties

Fortunately we are not completely on our own when writing constant-time programs. There are several approaches to automatically verify constant-time properties. Rodrigues, Pereira, and Aranha [43] use information flow analysis to find side-channels. Almeida et al. [6] developed a formal framework to capture timing side-channels as well as an automated verification tool, ctverif, based on the intermediate verification language Boogie.

A different approach is the design of domain specific programming languages like FaCT [15, 14] or Jasmin [5]. Both of these come with special compilers and verification infrastructure, that can automatically verify the compiled code, with respect to constant-time properties.

Another direction is related to fuzzing. He, Emmi, and Ciocarlie [22] use fuzzing to find constant-time issues, while Reparaz, Balasch, and Verbauwhede [41] use statistical methods to detect timing differences.

[14] example taken from Pascal Cuoq at https://trust-in-soft.com/when-in-doubt-express-intent-and-leave-the-rest-to-the-compiler/

Listing 2.5 – Ternary operator implemnted in several ways

```
1  int cond(int a, int b, int c) {
2    if (c) {
3      return a;
4    } else {
5      return b;
6    }
7  }
8
9  int ct_cond(int a, int b, int c) {
10   int t = a ^ b;
11   return b & ((-c) & t);
12 }
13
14 int ct_cond2(int a, int b, int c) {
15   int t = a ^ b;
16   c = !!c;
17   return b & ((-c) & t);
18 }
19
20 int ct_cond3(int a, int b, int c) {
21   c = !!c;
22   return (a & -c) | (b & ~-c);
23 }
```

Listing 2.6 – Ternary operator from Listing 2.5 compiled using clang (left) and gcc (right). Conditional move instructions highlighted in red.

```
cond:
testl %edx, %edx
cmovel  %esi, %edi
movl  %edi, %eax
retq

ct_cond:
xorl  %esi, %edi
negl  %edx
andl  %esi, %edx
andl  %edi, %edx
movl  %edx, %eax
retq

ct_cond2:
xorl  %esi, %edi
testl %edx, %edx
cmovel  %edx, %edi
andl  %esi, %edi
movl  %edi, %eax
retq

ct_cond3:
testl %edx, %edx
cmovel  %esi, %edi
movl  %edi, %eax
retq
```

```
cond:
testl %edx, %edx
movl  %esi, %eax
cmovne  %edi, %eax
ret

ct_cond:
movl  %edx, %eax
xorl  %esi, %edi
negl  %eax
andl  %esi, %eax
andl  %edi, %eax
ret

ct_cond2:
testl %edx, %edx
setne %al
xorl  %esi, %edi
movzbl  %al, %eax
negl  %eax
andl  %edi, %eax
andl  %esi, %eax
ret

ct_cond3:
testl %edx, %edx
setne %dl
movzbl  %dl, %edx
movl  %edx, %eax
subl  $1, %edx
negl  %eax
andl  %edx, %esi
andl  %edi, %eax
orl %esi, %eax
ret
```

Lastly, the dynamic analysis tool Valgrind [45] can be used to check some of the desired properties. Its tool Memcheck can track uninitialized memory and detect any branches and memory accesses depending on it. Langley developed a patch, called ctgrind [32], providing functions to instrument a program to mark memory locations as containing secret values. These are then treated by Memcheck as uninitialized, detecting such timing issues. Since Valgrind works by emulating the execution of the binary on a virtual processor, we are not only able to detect issues stemming from design problems in the higher-level language used, but also issues introduced by the compiler.

We are not completely safe, though. Valgrind only cares about branches and memory accesses, and not CPU instructions that are inherently non-constant-time. Additionally, Memcheck can report false positives, due to the way Valgrind simulates the CPU. This can be especially common when using Intel's vector instructions, since their state is not tracked at the same granularity as the normal instructions. Since compilers can use vector instructions on higher optimization levels this also increases the false positive rate on higher optimization levels [46, Section 2.2].

2.3.1. Conclusion

As we have seen in this section, constant-time programming requires a lot of care and knowledge of low-level details. Its correctness directly depends on aspects of the machine, like integer representation. Moreover we cannot rely on compiler support, in fact we cannot trust it. In the following chapters we will now use the knowledge presented here to identify issues in the NTS-KEM reference implementation (Chapter 3) and provide a solutions to these issues (Chapter 4).

3. NTS-KEM

In this chapter we will explain the design of NTS-KEM, and discuss its implementation and the timing issues that arise from it. We will not provide a detailed discussion of the security of NTS-KEM with regards to classical or quantum cryptographic notions. These can be found in the NTS-KEM specification [4] as well as in the QROM security proof by Maram [33]. We restrict ourselves to timing based side-channels.

3.1. Specification

This section is directly taken from section 3 of the NTS-KEM submission [4], with a few minor adjustments. NTS-KEM is a KEM based on the McEliece [34] and the Niederreiter [37] PKE scheme. As such NTS-KEM is based on error correcting codes, more specifically binary separable Goppa codes. Such a code $\mathcal{C}_\mathcal{G}$ is defined by a Goppa polynomial $G(z) = g_0 + g_1 z + \cdots + g_\tau z^\tau \in \mathbb{F}_{2^m}[z]$ fulfilling the following properties:

- $G(z)$ has no roots in $\mathbb{F}_{2^m}$;
- $G(z)$ has no repeated roots in any extension field, which guarantees that $\mathcal{C}_\mathcal{G}$ is capable of correcting up to τ errors.

It can be described by a generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$, or a parity-check matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k) \times n}$, such that $\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0}$. A vector $\mathbf{w} \in \mathbb{F}_2^k$ can be encoded as a codeword in $\mathcal{C}$ as $\mathbf{c} = \mathbf{w} \cdot \mathbf{G} \in \mathbb{F}_2^n$. Moreover, for any codeword $\mathbf{c}$ in $\mathcal{C}$, we have $\mathbf{c} \cdot \mathbf{H}^T = \mathbf{0}$. More generally, given any vector $\mathbf{v} \in \mathbb{F}_2^n$, the vector $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T \in \mathbb{F}_2^{n-k}$ is called a *syndrome*. The problem of syndrome decoding is to find a vector of minimum weight $\mathbf{v} \in \mathbb{F}_2^n$ such that $\mathbf{s} = \mathbf{v} \cdot \mathbf{H}^T$ given a syndrome $\mathbf{s}$.

We can construct a decoding algorithm that can correct up to τ errors as follows. Let $\mathbf{c}_{rec} = \mathbf{c} + \mathbf{e}$, where $\mathbf{c} = \mathbf{w} \cdot \mathbf{G}$. Then $s \coloneqq \mathbf{c}_{rec} \cdot \mathbf{H}^T = \mathbf{w} \cdot \mathbf{G}\mathbf{H}^T + \mathbf{e} \cdot \mathbf{H}^T = \mathbf{e} \cdot \mathbf{H}^T$. We can then compute $\mathbf{e}$, e.g by using a lookup table, which is precomputed for all possible error vectors $\mathrm{hw}(\mathbf{e}) \leq \tau$. This then allows recovering the original codeword $\mathbf{c} = \mathbf{c}_{rec} - \mathbf{e}$. Lastly we have to map the codeword $\mathbf{c}$ back to a message $\mathbf{w}$, i.e. we have to solve $\mathbf{w} \cdot \mathbf{G} = \mathbf{c}$ for $\mathbf{w}$.

An instance of NTS-KEM is characterized by four public parameters:

- $n = 2^m$: a power-of-two, positive integer, which denotes the length of codewords.

- τ: a positive integer denoting the number of errors corrected by the code.

- $f(x)$: an irreducible polynomial of degree m over $\mathbb{F}_2$, defining the extension field $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$.

- $\ell = 256$: a positive integer, which denotes the length of the random key to be encapsulated.

Furthermore it is required that $\log \binom{n}{\tau} \geq \ell$, and $\ell < k < n$, where $k = n - \tau m$.

NTS-KEM makes use of a pseudorandom bit generator to produce ℓ-bit binary strings, which we denote by $H_\ell(\cdot)$. As pseudorandom bit generator NTS-KEM uses SHA3-256.

3.1.1. Key-Generation

The procedure to generate a NTS-KEM key-pair is as follows:

1. Randomly generate a monic Goppa polynomial of degree τ

$$G(z) = g_0 + g_1 z + \cdots + g_{\tau-1} z^{\tau-1} + z^\tau,$$

where $g_i \in \mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$, with $g_0 \neq 0$. The polynomial $G(z)$ defines a binary Goppa code $\overline{\mathcal{C}_\mathcal{G}}$ of length $n = 2^m$, dimension $k = n - \tau m$, capable of correcting up to τ errors.

2. Randomly generate a permutation vector $\mathbf{p}$ of length n, representing a permutation $\pi_\mathbf{p}$ on the set of n elements.

3. Construct a generator matrix in the reduced row echelon form $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}]$ of a *permuted* code $\mathcal{C}_\mathcal{G}$ as follows:

   a) Let $\beta \in \mathbb{F}_{2^m}$ be a root of $f(x)$, where $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/f(x)$ and B be a basis of $\mathbb{F}_{2^m}$, $B = \langle \beta^{(m-1)}, \ldots, \beta, 1 \rangle$. The i-th element of $\mathbb{F}_{2^m}$ in the basis of B is defined by

$$B[i] = \{b_0\beta^{(m-1)} + b_1\beta^{(m-2)} + \ldots + b_{m-2}\beta + b_{m-1} \ : \ b_j \in \{0,1\}\},$$

   where $i = \sum_{j=0}^{m-1} b_j 2^j$.

   b) Let $\mathbf{a}'$ be the sequence of all elements of $\mathbb{F}_{2^m}$ given by

$$\mathbf{a}' = (a_0, a_1, a_2, \ldots, a_{n-2}, a_{n-1}) = (B[0], B[1], B[2], \ldots, B[n-2], B[n-1]),$$

   where $B[i]$ is defined above.

   c) Let $\mathbf{a} = \pi_\mathbf{p}(\mathbf{a}') = (a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}}) \in \mathbb{F}_{2^m}^n$ be the sequence obtained by re-ordering the elements of $\mathbf{a}'$ according to $\pi_\mathbf{p}$.

d) Construct the parity-check matrix $\mathbf{H}_m \in \mathbb{F}_{2^m}^{\tau \times n}$ using the sequence $\mathbf{a}$, as described by

$$\mathbf{H}_m = \begin{bmatrix} G(a_0)^{-2} & G(a_1)^{-2} & \cdots & G(a_{n-1})^{-2} \\ a_0 G(a_0)^{-2} & a_1 G(a_1)^{-2} & \cdots & a_{n-1} G(a_{n-1})^{-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_0^{\tau-1} G(a_0)^{-2} & a_1^{\tau-1} G(a_1)^{-2} & \cdots & a_{n-1}^{\tau-1} G(a_{n-1})^{-2} \end{bmatrix}.$$

Let $\mathbf{h} = (h_{p_0}, h_{p_1}, \ldots, h_{p_{n-1}}) \in \mathbb{F}_{2^m}^n$ be the first row of $\mathbf{H}_m$.

e) Let $B(a_i) = (b_{i0}, b_{i1}, \ldots, b_{i(m-1)})$ be a representation of a_i over $\mathbb{F}_2$, i.e.

$$a_i = b_{i0} + b_{i1}\beta + b_{i2}\beta^2 + \cdots + b_{i(m-1)}\beta^{m-1},$$

where $b_{ij} \in \mathbb{F}_2$. Transform $\mathbf{H}_m$ to $\mathbf{H} \in \mathbb{F}_2^{m\tau \times n}$ using operator $B(\cdot)^T$.

f) Transform $\mathbf{H}$ to reduced row echelon form, re-ordering its columns if necessary, such that the identity matrix $\mathbf{I}_{n-k}$ occupies the last $n-k$ columns of $\mathbf{H}$. If ρ is the permutation representing this re-ordering of columns, apply the same re-ordering to the vectors $\mathbf{a}$, $\mathbf{h}$ and $\mathbf{p}$, i.e. make $\mathbf{a} = \rho(\mathbf{a})$, $\mathbf{h} = \rho(\mathbf{h})$ and $\mathbf{p} = \rho(\mathbf{p})$.

g) Construct the generator matrix $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}] \in \mathbb{F}_2^{k \times n}$ of the permuted code $\mathcal{C}_\mathcal{G}$ from the parity-check matrix $\mathbf{H} = [\mathbf{Q}^T \mid \mathbf{I}_{n-k}]$.

4. Sample $\mathbf{z} \in \mathbb{F}_2^\ell$ uniformly at random.

5. Partition the vectors $\mathbf{a}$ and $\mathbf{h}$ as $\mathbf{a} = (\mathbf{a}_a \mid \mathbf{a}_b \mid \mathbf{a}_c)$ and $\mathbf{h} = (\mathbf{h}_a \mid \mathbf{h}_b \mid \mathbf{h}_c)$, where $\mathbf{a}_a, \mathbf{h}_a \in \mathbb{F}_{2^m}^{k-\ell}$, $\mathbf{a}_b, \mathbf{h}_b \in \mathbb{F}_{2^m}^\ell$ and $\mathbf{a}_c, \mathbf{h}_c \in \mathbb{F}_{2^m}^{n-k}$. Finally, define

$$\mathbf{a}^* = (\mathbf{a}_b \mid \mathbf{a}_c) \quad \text{and} \quad \mathbf{h}^* = (\mathbf{h}_b \mid \mathbf{h}_c).$$

The NTS-KEM public and private keys are given as follows.

- The public key is $\mathrm{pk} = (\mathbf{Q}, \tau, \ell)$, where $\mathbf{Q} \in \mathbb{F}_2^{k \times (n-k)}$ and τ, ℓ are positive integers (determined in the parameter sets).
- The private key is $\mathrm{sk} = (\mathbf{a}^*, \mathbf{h}^*, \mathbf{p}, \mathbf{z}, \mathrm{pk})$, where $\mathbf{a}^*, \mathbf{h}^* \in \mathbb{F}_{2^m}^{n-k+\ell}$, $\mathbf{p} \in \mathbb{F}_{2^m}^n$ and $\mathbf{z} \in \mathbb{F}_2^\ell$.

3.1.2. Encapsulation

Given a NTS-KEM public key $\mathrm{pk} = (\mathbf{Q}, \tau, \ell)$, the encapsulation process produces two vectors over $\mathbb{F}_2$, one of which is a random vector $\mathbf{k}_r$, where $|\mathbf{k}_r| = \ell = 256$; the other is the ciphertext $\mathbf{c}^*$ encapsulating $\mathbf{k}_r$.

NTS-KEM encapsulation makes use of the following function, acting on input $(\mathrm{pk}, \mathbf{e})$ and denoted as $\textsc{Encap}(\mathrm{pk}, \mathbf{e})$, which proceeds as follows.

1. Partition $\mathbf{e}$ as $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$, where $\mathbf{e}_a \in \mathbb{F}_2^{k-\ell}$, $\mathbf{e}_b \in \mathbb{F}_2^{\ell}$ and $\mathbf{e}_c \in \mathbb{F}_2^{n-k}$.

2. Compute $\mathbf{k}_e = H_\ell(\mathbf{e}) \in \mathbb{F}_2^{\ell}$.

3. Construct the message vector $\mathbf{m} = (\mathbf{e}_a \mid \mathbf{k}_e) \in \mathbb{F}_2^{k}$.

4. Perform systematic encoding of $\mathbf{m}$ with $\mathbf{Q}$:

$$\begin{aligned}\mathbf{c} &= (\mathbf{m} \mid \mathbf{m} \cdot \mathbf{Q}) + \mathbf{e} \\ &= (\mathbf{e}_a \mid \mathbf{k}_e \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q}) + (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{k}_e + \mathbf{e}_b \mid (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c) \\ &= (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)\,,\end{aligned}$$

   where $\mathbf{c}_b = \mathbf{k}_e + \mathbf{e}_b$ and $\mathbf{c}_c = (\mathbf{e}_a \mid \mathbf{k}_e) \cdot \mathbf{Q} + \mathbf{e}_c$. Then remove the first $k - \ell$ coordinates (all zero) from $\mathbf{c}$ to obtain $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c) \in \mathbb{F}_2^{n-k+\ell}$.

5. Output the pair $(\mathbf{k}_r, \mathbf{c}^*)$ where $\mathbf{k}_r = H_\ell(\mathbf{k}_e \mid \mathbf{e}) \in \mathbb{F}_2^{\ell}$.

NTS-KEM encapsulation is then defined as:

1. Generate uniformly at random an error vector $\mathbf{e} \in \mathbb{F}_2^{n}$ with Hamming weight τ.

2. Call $\textsc{Encap}(\mathrm{pk}, \mathbf{e})$ and return its outputs.

3.1.3. Decapsulation

The decapsulation of a NTS-KEM ciphertext $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c)$ proceeds as follows:

1. Consider the vector $\mathbf{c} = (\mathbf{0}_a \mid \mathbf{c}_b \mid \mathbf{c}_c) \in \mathbb{F}_2^{n}$, and apply a decoding algorithm — using the private key — to recover a permuted error pattern $\mathbf{e}'$.

2. Compute the error vector $\mathbf{e} = \pi_{\mathbf{p}}(\mathbf{e}')$ and partition $\mathbf{e} = (\mathbf{e}_a \mid \mathbf{e}_b \mid \mathbf{e}_c)$.

3. Compute $(\mathbf{k}_r, \mathbf{c}') = \textsc{Encap}(\mathrm{pk}, \mathbf{e})$. Verify that $\mathbf{c}' = \mathbf{c}^*$ and $\mathrm{hw}(\mathbf{e}) = \tau$. If yes, return $\mathbf{k}_r \in \mathbb{F}_2^{\ell}$; otherwise return $H_\ell(\mathbf{z} \mid \mathbf{1}_a \mid \mathbf{c}_b \mid \mathbf{c}_c)$.

3.2. Implementation and Issues

For each of the three core functions of NTS-KEM we will now go through the steps in the specification and explain, in as much detail as needed, how the optimized implementation implements them. Moreover we will identify timing issues as we go along.

3.2.1. Key-Generation

Step 1 starts with generating a random Goppa polynomial $G(z)$. This is done by first sampling random coefficients and then checking that the resulting polynomial is a valid Goppa polynomial. The Goppa polynomial is represented as an array of coefficients, stored in 16-bit integers where the most significant $16 - \tau$ bit are zero, together with some book-keeping information (e.g. the degree). Sampling is straight-forward, just sample m bits for every coefficient. Note that we want a monic polynomial. Thus the first coefficient is always 1 and we only need to sample τ coefficients. The validity check is then done in three steps:

1. Checking that $G(z)$ has no roots in $\mathbb{F}_{2^m}$. This is done using an additive FFT [18]. NTS-KEM makes use of a bitsliced implementation, which is naturally constant-time.

2. Computing $\frac{d}{dz}G(z)$. Note that computing the derivative of a polynomial in a field of characteristic 2 results in a polynomial with only even coefficients. More precisely the derivative is given by $\frac{d}{dz}G(z) = \sum_{j=0}^{\lceil \tau/2 \rceil - 1} g_{2j+1} z^{2j}$. The implementation therefore simply copies every uneven coefficient of $G(z)$ to the next smaller array index, and ensures that all other array elements are zero. Since (now leading) coefficients could be zero, the degree of the derivative is computed. This was done in the following way

   ```
   for (i=0; i<fx->degree; ++i) {
     /* in case coefficients are zero */
     if (dx->coeff[fx->degree-1-i])
       break;
     --dx->degree;
   }
   ```

   which leads to a timing issue.

   Issue 1 (Computing the derivative of a polynomial)**.** *Degree computation in formal derivative potentially leaks the number of consecutive zero coefficients following the leading coefficient in the secret polynomial $G(z)$.*

3. Checking that $\mathrm{GCD}\left(G(z), \frac{d}{dz}G(z)\right) = 1$. This ensures $G(z)$ has no repeated roots in any extension field. This is achieved using a version of Euclid's algorithm for polynomials (see Algorithm 3.1). However, Euclid's algorithm is a variable time algorithm.

   Issue 2 (Computing the GCD of two polynomials)**.** *The number of loop iterations in Algorithm 3.1 depends on the inputs. Moreover the polynomial modular reduction has to be implemented in constant-time if needed.*

Step 2 of key-generation is generating the permutation vector uniformly at random. This is done using the Fisher-Yates shuffle depicted in Algorithm 3.2 on

Algorithm 3.1 Greatest common divisor of $a(z)$ and $b(z)$. Algorithm 6 in the NTS-KEM submission [4].

1: **function** GCD($a(z), b(z)$)
Require: $\deg a(z) \geq \deg b(z)$
2: **while** $\deg b(z) \geq 0$ **do**
3: $t(z) \leftarrow b(z)$
4: $b(z) \leftarrow a(z) \mod b(z)$
5: $a(z) \leftarrow t(z)$
6: **end while**
7: **return** $a(z)$
8: **end function**

an input $(0, 1, ..., n-1)$. While the loop of the Fisher-Yates shuffle has a constant bound that is not secret (it only depends on n, a public parameter), it contains secret dependent memory accesses.

Issue 3 (Generating random permutations)**.** *The memory accesses in Line 5 of Algorithm 3.2 depend on the randomness sampled in Line 4, which is a secret.*

Algorithm 3.2 Fisher-Yates shuffle on sequence $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$. Algorithm 7 in the NTS-KEM submission [4].

1: **function** RandomShuffle($\mathbf{a}$)
2: $i \leftarrow n - 1$
3: **while** $i > 0$ **do**
4: $r \leftarrow_{\$} \{0, 1, \ldots, i\}$
5: Swap a_i with a_r
6: $i \leftarrow i - 1$
7: **end while**
8: **return** the shuffled sequence $\mathbf{a}$
9: **end function**

To sample the randomness in Line 4 the Knuth-Yao algorithm (Algorithm 3.3) is used. Unfortunately the Knuth-Yao algorithm is not constant-time.

Issue 4 (Sampling bounded integers)**.** *The number of iterations of the Knuth-Yao algorithm (Algorithm 3.3) depends on the sampled randomness, which is a secret.*

Step 3 starts with initializing $\mathbf{a}$ as in steps 3a and 3b, involving no secrets. Then we compute a vector $\bar{\mathbf{h}} = (G(B[0]), G(B[1]), \ldots, G(B[n-1]))$ using the bitsliced additive FFT. Both $\mathbf{a}$ and $\bar{\mathbf{h}}$ are then permuted using $\mathbf{p}$.

Issue 5 (Permuting elements of $\mathbb{F}_{2^m}$)**.** *Permuting* $\mathbf{a}$ *and* $\bar{\mathbf{h}}$ *uses* $\mathbf{p}$ *as array index, but* $\mathbf{p}$ *is secret.*

Algorithm 3.3 Knuth-Yao algorithm to generate a uniform random number in $\{0, \ldots, i-1\}$. Algorithm 8 in the NTS-KEM submission [4].

```
1:  function KnuthYaoUniformRNG(i)
2:      u ← 1
3:      x ← 0
4:      while TRUE do
5:          while u < i do
6:              u ← 2u
7:              x ← 2x + RandomBit
8:          end while
9:          d ← u − i
10:         if x ≥ d then
11:             return x − d
12:         else
13:             u ← d
14:         end if
15:     end while
16: end function
```

We then compute $\mathbf{h} = \bar{\mathbf{h}}^{-2}$, and then $\mathbf{H}_m$ as in step 3d from $\mathbf{h}$ and $\mathbf{a}$ using constant-time procedures for inversion, squaring, and multiplication in $\mathbb{F}_{2^m}$ described in appendix A of the NTS-KEM submission [4] which implicitly also perform the transformation from step 3e.

The transformation into reduced row echelon form (Step 3f) is then implemented using an efficient implementation of the Gauss-Jordan algorithm by Albrecht, Bard, and Pernet [2] based on the method of four Russians (M4R). Unfortunately this algorithm is not constant time.

Issue 6 (Gaussian elimination)**.** *The M4R-based Gaussian elimination is designed to utilize the structure of the input matrix to achieve higher performance for most inputs, but can degrade to the performance of classical Gauss-Jordan elimination in the worst case. This results in an inherently non-constant-time algorithm.*

After the reduced row echelon transform, the columns of the matrix are permuted to ensure that the identity matrix $\mathbf{I}_{n-k}$ occupies the last $n-k$ columns. This permutation is done in a non-constant-time fashion.

Issue 7 (Permuting matrix columns)**.** *Permuting the matrix columns is done by linearly searching the row for a nonzero element. The corresponding columns are then swapped and the permutation is mirrored onto* $\mathbf{h}$ *and* $\mathbf{a}$*. These swaps use the column indices as array indices leading to a potential cache side-channel.*

For step 3g the matrix is transposed and truncated by iterating over the matrix. This is done in constant time.

Listing 3.1 – Excerpt from the matrix multiplication corresponding to multiplication of $\mathbf{e}_a$ with $\mathbf{Q}$, taken from the NTS-KEM reference implementation [3].

```
1    \\ A and R are constants corresponding to the size of the
        ↪ arrays chunks
2    for (i=0; i<A; i++) {
3      memcpy(&v, &e[i*sizeof(v)], sizeof(v));
4      while (v) {
5        l = (int32_t)lowest_bit_idx(v);
6        v ^= (ONE << l);
7        l += (BITSIZE*i);
8        for (j=0; j<R; j++) {
9          c_c[j] ^= Q[l][j];
10       }
11     }
12   }
```

Finally we sample $\mathbf{z}$ using a secure PRNG and copy $\mathbf{a}^*$ and $\mathbf{h}^*$ into the appropriate return variables, completing steps 4 and 5 as well as key-generation. Before the keys are returned however, there is an additional last step, where the keys are serialized, to ensure they are in a well-defined platform independent format.

3.2.2. Encapsulation

Encapsulation consists of two steps, generation of $\mathbf{e}$ uniformly at random, and the core encapsulation procedure. Generating $\mathbf{e}$ so that $\mathrm{hw}(e) = \tau$ is done by using the Fisher-Yates shuffle (Algorithm 3.2) on the input $(0, ..., 0, 1..., 1)$.

Issue 8 (Sampling a random error vector)**.** *As explained in issue 3, the Fisher-Yates shuffle is not constant-time.*

The core encapsulation procedure essentially consists of computing hash functions and performing the encoding via matrix multiplication. To compute the hash functions in steps 2 and 5 NTS-KEM uses *Keccak-Tiny* [21], which is constant-time. The matrix multiplication is done in a vectorized way using 64-bit integer xor. Moreover the matrix multiplication is optimized to skip unnecessary xor operations, as can be seen from Listing 3.1.

Issue 9 (matrix vector multiplication over $\mathbb{F}_2$)**.** *The loop in Line 4 of Listing 3.1 depends on the Hamming weight of* $\mathbf{e}_a$ *(via the variable* `v`*).*

Issue 10 (`lowest_bit_idx`)**.** *The function* `lowest_bit_idx` *used in Listing 3.1 and shown in Listing 3.2 is branching on its (secret) input.*

Listing 3.2 – `lowest_bit_idx` as used in Listing 3.1.

```
1  uint64_t lowest_bit_idx(uint64_t x)
2  {
3    uint64_t r = 0;
4    if (!x) return -1;
5    x &= -x;
6
7    if ( x & 0xffffffff00000000UL ) r += 32;
8    if ( x & 0xffff0000ffff0000UL ) r += 16;
9    if ( x & 0xff00ff00ff00ff00UL ) r +=  8;
10   if ( x & 0xf0f0f0f0f0f0f0f0UL ) r +=  4;
11   if ( x & 0xccccccccccccccccUL ) r +=  2;
12   if ( x & 0xaaaaaaaaaaaaaaaaUL ) r +=  1;
13
14   return r;
15 }
```

3.2.3. Decapsulation

The first step of decapsulation is the decoding step. Decoding is implemented based on the Berlekamp-Massey algorithm. For this, we first have to compute the 2τ syndromes of the ciphertext $\mathbf{c}^*$. This is done using the Algorithm 3.4, which is implemented in constant-time by using constant-time multiplication and addition implementations (note that all bounds in the sums and the loop are public parameters). Then we apply the Berlekamp-Massey algorithm (see algorithm 3 of

Algorithm 3.4 Syndrome Computation on ciphertext $\mathbf{c}^* = (\mathbf{c}_b \mid \mathbf{c}_c)$. Algorithm 2 in the NTS-KEM submission [4].

function ComputeSyndrome($\mathbf{c}^*, \mathbf{a}^*, \mathbf{h}^*$)
Require: $\mathbf{c}^* \leftarrow (c_{b,0}, c_{b,1}, \ldots, c_{b,\ell-1} \mid c_{c,0}, c_{c,1}, \ldots, c_{c,r-1})$
Require: $\mathbf{a}^* \leftarrow (a_{b,0}, a_{b,1}, \ldots, a_{b,\ell-1} \mid a_{c,0}, a_{c,1}, \ldots, a_{c,r-1})$
Require: $\mathbf{h}^* \leftarrow (h_{b,0}, h_{b,1}, \ldots, h_{b,\ell-1} \mid h_{c,0}, h_{c,1}, \ldots, h_{c,r-1})$
    $s_0 \leftarrow \sum_{j=0}^{\ell-1}(c_{b,j} \cdot h_{b,j}) + \sum_{j=0}^{r-1}(c_{c,j} \cdot h_{c,j})$
    $i \leftarrow 1$
    while $i < 2\tau$ **do**
        $\mathbf{h}^* \leftarrow \mathbf{h}^* \cdot \mathbf{a}^*$ ▷ Pointwise component multiplication modulo $\mathbb{F}_{2^m}$
        $s_i \leftarrow \sum_{j=0}^{\ell-1}(c_{b,j} \cdot h_{b,j}) + \sum_{j=0}^{r-1}(c_{c,j} \cdot h_{c,j})$
        $i \leftarrow i + 1$
    end while
    return $\mathbf{s} = (s_0, s_1, \ldots, s_{2\tau-1})_{\mathbb{F}_{2^m}}$
end function

the NTS-KEM submission [4]) using a bitsliced implementation. While a bitsliced

Listing 3.3 – bit scan reverse. Finds index of highest set bit.

```
1  static inline uint64_t bsr(uint64_t x) {
2    uint64_t r = 0;
3    if (x & 0xffffffff00000000UL )  { x >>= 32;  r += 32; }
4    if (x & 0xffff0000UL) { x >>= 16; r += 16; }
5    if (x & 0x0000ff00UL) { x >>=  8; r +=  8; }
6    if (x & 0x000000f0UL) { x >>=  4; r +=  4; }
7    if (x & 0x0000000cUL) { x >>=  2; r +=  2; }
8    if (x & 0x00000002UL) {           r +=  1; }
9
10   return mux(isZero(x), 0, r);
11 }
```

implementation is usually inherently constant-time, this implementation makes use of Listing 3.3, to find the index of the highest set bit.

Issue 11 (Bit scan reverse)**.** *`bsr` in Listing 3.3 is not constant-time due to branching.*

Additionally there were several instances where logical or (`||`) was used.

Issue 12 (Logical boolean operators)**.** *Usage of logical or to compute the disjunction of two binary values in the implementation of the Berlekamp-Massey algorithm.*

The next step of decoding evaluates an error locator polynomial $\sigma(x)$ returned by the Berlekamp-Massey algorithm using an additive FFT. The additive FFT is done using a bitsliced implementation without any timing issues. From the evaluation of $\sigma(x)$ we then know the positions of the errors and can therefore recover $\mathbf{e}'$, concluding the decoding step.

Next we have to permute the recovered error e' according to $\mathbf{p}$.

Issue 13 (Permuting bit vectors)**.** *Similarly to issue 5,* $\mathbf{p}$ *is used as index to directly access the array containing the vector, but* $\mathbf{p}$ *is secret.*

The final step is recomputing Encap(pk, $\mathbf{e}$), which is coverd in Section 3.2.2, and performing the verification as in step 3. Here care is taken to ensure that both possible return values are computed and the final return value is chosen using a mux, to not create a timing issue in case of errors.

4. Constant-Time NTS-KEM

In Chapter 3 we identified 13 issues with regards to timing based side-channels in the current NTS-KEM implementation:

1. Computing the derivative of a polynomial
2. Computing the GCD of two polynomials
3. Generating random permutations
4. Sampling bounded integers
5. Permuting elements of $\mathbb{F}_{2^m}$
6. Gaussian elimination
7. Permuting matrix columns
8. Sampling a random error vector
9. matrix vector multiplication over $\mathbb{F}_2$
10. `lowest_bit_idx`
11. Bit scan reverse
12. Logical boolean operators
13. Permuting bit vectors

We will first show how to fix the smaller issues, namely issues 1 and 9 to 12. We then discuss the Gaussian elimination and matrix column swaps in Section 4.2 (issues 6 and 7). Section 4.5 shows how to sample the random objects we need (issues 3 and 4). In Section 4.3 we present the constant-time GCD algorithm we used (issue 2) and finally in Section 4.4 we discuss how to perform permutations in constant time (issues 5 and 13).

4.1. Fixing the Smaller Issues

We start by taking care of the smaller issues. Issue 12 is trivial. The involved operations are always on operands that are either 0 or 1 and can thus just be replaced by the corresponding bitwise operations. The remaining issues are matrix multiplication and simple issues involving branching. For the branching issues, two are due to conditionals, and one because of early loop termination.

Matrix Multiplication over $\mathbb{F}_2$

To make the matrix multiplication constant-time we simply do not skip the cases where bits are zero. Instead we simply iterate over all bits and perform a plain and straight-forward matrix multiplication as shown in Listing 4.1. As a side effect we also solve issue 10, since `lowest_bit_index` is not used anymore.

Listing 4.1 – Excerpt of constant-time matrix vector multiplication over $\mathbb{F}_2$. (Compare with Listing 3.1)

```
1  for (i=0; i<A; i++) {
2    memcpy(&v, &e[i*sizeof(v)], sizeof(v));
3    for(l = 0; l < sizeof(v)<<3; l++) {
4      for (j=0; j<R; j++) {
5        c_c[j] ^= -((v & (ONE << l))>>l) & Q[l + (BITSIZE*i)][j];
6      }
7    }
8  }
```

Listing 4.2 – Constant-time degree computation.

```
1  int zero = 1;
2  for (i = 0; i < fx->degree; i++) {
3    zero &= isZero(dx->coeff[fx->degree-1-i]);
4    dx->degree -= zero;
5  }
```

Computing the Degree in Polynomial Derivative

We have to remove the early break in issue 1. We can achieve this by storing the value we subtract from the degree in a variable, which we set to 0 once we have found a nonzero coefficient. This is shown in Listing 4.2.

Bit Scan

Issues 10 and 11 are basically the same issue. The functions `lowest_bit_index` and `bsr` (see Listings 3.2 and 3.3) consist of a sequence of branches. Issue 10 was already solved above, so we only need to focus on issue 11. Since every branch only consists of very few computations, we can easily fix this by always performing the computations and then either using the results or keeping the old values using a mux, as we have seen in Section 2.3. Listing 4.3 shows a constant-time version of `bsr`.

Listing 4.3 – Constant-time bit scan reverse.

```
1   uint64_t bsr(uint64_t x) {
2     uint64_t r = 0;
3     uint64_t y, t, ctrl;
4     // if (x & 0xffffffff00000000UL )  { x >>= 32;  r += 32; }
5     y = x >> 32;
6     t = r + 32;
7     ctrl = isZero(x & 0xffffffff00000000UL);
8     r = mux(ctrl, r, t);
9     x = mux(ctrl, x, y);
10    // if (x & 0xffff0000UL) { x >>= 16; r += 16; }
11    y = x >> 16;
12    t = r + 16;
13    ctrl = isZero(x & 0xffff0000UL);
14    r = mux(ctrl, r, t);
15    x = mux(ctrl, x, y);
16    // if (x & 0x0000ff00UL) { x >>=  8; r +=  8; }
17    y = x >> 8;
18    t = r + 8;
19    ctrl = isZero(x & 0x0000ff00UL);
20    r = mux(ctrl, r, t);
21    x = mux(ctrl, x, y);
22    // if (x & 0x000000f0UL) { x >>=  4; r +=  4; }
23    y = x >> 4;
24    t = r + 4;
25    ctrl = isZero(x & 0x000000f0UL);
26    r = mux(ctrl, r, t);
27    x = mux(ctrl, x, y);
28    // if (x & 0x0000000cUL) { x >>=  2; r +=  2; }
29    y = x >> 2;
30    t = r + 2;
31    ctrl = isZero(x & 0x0000000cUL);
32    r = mux(ctrl, r, t);
33    x = mux(ctrl, x, y);
34    // if (x & 0x00000002UL) {           r +=  1; }
35    t = r + 1;
36    ctrl = isZero(x & 0x00000002UL);
37    r = mux(ctrl, r, t);
38
39    return mux(isZero(x), 0, r);
40  }
```

4.2. Gaussian Elimination

NTS-KEM uses codes in systematic form, i.e. with a generator matrix of the form $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{Q}]$. Remember that for NTS-KEM we have an $m \times n$ full rank matrix, with $m < n$ (note that in this section m and n do not refer to the NTS-KEM parameters), which we want to transform into the above form. Unfortunately a full Gauss-Jordan elimination runs in $\mathcal{O}(m^2 \cdot n)$ for such a matrix.

We start by shortly repeating how Gauss-Jordan elimination works. We then show how to implement a constant time version of Gauss-Jordan elimination in Section 4.2.2. Since a full Gauss-Jordan elimination is slow, we also show a alternative version in Section 4.2.3, which increases performance significantly at the price of a higher usage of randomness.

4.2.1. Plain Gaussian Elimination

Algorithm 4.1 Gauss-Jordan elimination

Require: $\mathbf{M}$ is a $m \times n$ matrix.
Ensure: $\mathbf{M}$ is in reduced row echelon form.

```
1:  procedure GaussJordan(M)
2:      r = 0
3:      while r < m do
4:          for c ∈ (0, 1, ..., n − 1) do
5:              Find r' ≥ r s.t. M_{r',c} = 1.
6:              if found r' then
7:                  Swap rows r and r'
8:                  for all r' ∈ {0, 1, ..., m − 1} \ {r} with M_{r',c} = 1 do
9:                      Add row r to row r'
10:                 end for
11:                 r = r + 1
12:             end if
13:         end for
14:     end while
15: end procedure
```

First, remember the definition of the reduced row echelon form (rref). A matrix $\mathbf{M}$ is in rref if

- all non-zero rows are above all zero rows,
- the first non-zero entry in each non-zero row is 1,
- each column containing a leading 1, contains only this one non-zero value,
- each leading coefficient of a row is to the left of all leading coefficients of the rows below it.

Note that we are working on matrices over $\mathbb{F}_2$, so all values are either zero or one and addition is the same as subtraction Furthermore in the following we assume that $\mathbf{M}$ is an $m \times n$-matrix with $m < n$. We can then transform any such matrix $\mathbf{M}$ into rref, using the Gauss-Jordan elimination, which utilizes swapping of rows and replacing a row with the its sum with another row, i.e. $\mathbf{M}_i = \mathbf{M}_i + \mathbf{M}_j$. Pseudocode for the procedure is depicted in Algorithm 4.1.

4.2.2. Constant-Time Gaussian Elimination

We first discuss the constant-time Gauss-Jordan elimination, which transforms a matrix into reduced row echelon form. Then we will discuss how to permute the columns of the rref matrix to get a systematic matrix as defined above.

Constant-Time Gauss-Jordan Elimination

Gauss-Jordan elimination has many pitfalls for timing issues. Since our matrix has rank m it may happen that a column becomes zero during elimination. However we cannot use this for a speedup, since that might leak, especially via the memory access pattern, which columns become zero. Similarly we cannot stop when we have an rref matrix, since this leaks how many of the columns were needed, i.e. it leaks information about the number of columns that became zero during elimination. Moreover we may not leak which row we are currently working on, as this again leaks information about the zero columns, since we stay on the same row when encountering a zero column. Thus the timing between accessing new rows depends on the zero columns. (Note that ctgrind was actually able to find all of the above issues in a non-constant-time Gauss-Jordan elimination.)

Since we are working with a matrix over $\mathbb{F}_2$, multiplication and addition are simply bitwise *and* and *xor*. Our implementation is thus vectorized using 64-bit integers (and 256-bit avx registers in the avx version) to access chunks of the matrix rows. Algorithm 4.2 shows pseudocode for a constant-time Gauss-Jordan elimination. All operations that only involve vectors can be parallelized. We will now go over the algorithm and explain the important steps. Note that our implementation starts elimination in the lower right corner.

First we have to iterate over all columns c, for this we keep track of the row we are working on. As first step of the loop body, we copy the current row into a fixed area, by iterating over all rows. This ensures that across all loop iterations, accesses to the row currently worked on go to the same memory locations. We thus avoid leaking how many (column-)loop iterations we spend on the same row.

Next, we look for a pivot element (lines 10 to 27). We do this by swapping **currentRow** with $\mathbf{A}_r$ if $(A)_{r,c}$ while iterating over all rows r. Note that this reorders the rows wildly and in the end the row-th row appears twice in the matrix. The row that is missing is only stored in **currentRow**. Since columns can be all zeros, we also save whether we found a pivot.

The elimination step (lines 20 to 27) then performs two tasks, while iterating over all rows. Firstly, it writes **currentRow** to the *row*-th row, finalizing the row swapping started during pivot search. Secondly for all other rows r it adds (via xor) **currentRow** to them if $\mathbf{A}_{r,c}$ is set.

In the end, we update the *row* variable. We use the fact that the row variable will be negative once we finished elimination to be able to continue iterating over the columns. Lines 12 and 21 ensure that the writes occurring in the remaining iterations do not change the matrix. We visualize row swaps and eliminations on a simple example.

Example 4.2.1. Let $m = 3$ and $\mathbf{A} = \begin{pmatrix} \cdots & 1 & 1 & 1 \\ \cdots & 1 & 0 & 1 \\ \cdots & 0 & 1 & 1 \end{pmatrix}$. We will show the first iteration of the outermost loop (i.e. $c = n - 1$). We set $row = 2$ and $\mathbf{currentRow} = (\cdots, 0, 1, 1)$. We then start the pivot search loop (Line 10)

1. Then the first iteration of pivot search ($r = 2$) sets $found = 1$, but does not swap anything.
2. For $r = 1$ we then set $\mathbf{currentRow} = (\cdots, 1, 0, 1)$ and $\mathbf{A} = \begin{pmatrix} \cdots & 1 & 1 & 1 \\ \cdots & 0 & 1 & 1 \\ \cdots & 0 & 1 & 1 \end{pmatrix}$.
3. For $r = 0$ we set $\mathbf{currentRow} = (\cdots, 1, 1, 1)$ and $\mathbf{A} = \begin{pmatrix} \cdots & 1 & 0 & 1 \\ \cdots & 0 & 1 & 1 \\ \cdots & 0 & 1 & 1 \end{pmatrix}$.

The elimination loop (Line 20) then proceeds as follows.

1. For $r = 2$ we write **currentRow** to $\mathbf{A}_r$, finishing the pivot swap; $\mathbf{A} = \begin{pmatrix} \cdots & 1 & 0 & 1 \\ \cdots & 0 & 1 & 1 \\ \cdots & 1 & 1 & 1 \end{pmatrix}$.
2. For $r = 1$ and $r = 0$ we perform elimination by setting $\mathbf{A}_r = \mathbf{A}_r \otimes \mathbf{currentRow}$; $\mathbf{A} = \begin{pmatrix} \cdots & 0 & 1 & 0 \\ \cdots & 1 & 0 & 0 \\ \cdots & 1 & 1 & 1 \end{pmatrix}$.

We then continue with the next column. Here, repeating the above steps will result in swapping rows 1 and 0 and adding them to row 3, resulting in $\mathbf{A} = \begin{pmatrix} \cdots & 1 & 0 & 0 \\ \cdots & 0 & 1 & 0 \\ \cdots & 0 & 0 & 1 \end{pmatrix}$, which is in reduced row echelon form.

Algorithm 4.2 Constant-time Gauss-Jordan elimination.

Require: $\mathbf{A}$ is an $m \times n$ matrix.
Ensure: $\mathbf{A}$ is in reduced row echelon form.

```
 1: procedure RREF(A)
 2:     currentRow ← (0, ..., 0)
 3:     row ← m − 1
 4:     for c ∈ (n − 1, n − 2, ..., 0) do
 5:         for r ∈ (m − 1, m − 2, ..., 0) do          ▷ Select currentRow in ct
 6:             eq ← isEqual(r, row)
 7:             currentRow ← mux(eq, A_r, currentRow)
 8:         end for

 9:         found ← 0
10:         for r ∈ (m − 1, m − 2, ..., 0) do          ▷ Search for pivot
11:             m ←lessOrEquals(r, row)
12:             m ← m ∧biggerOrEqualsZero(row)
13:             m ← m ∧ A_{r,c}
14:             found ← found ∨ m
15:             mask ← (m, ..., m)

16:             t ← currentRow ⊗ A_r
17:             currentRow ← currentRow ⊗(t ∧ mask)
18:             A_r ← A_r ⊗(t ∧ mask)
19:         end for

20:         for r ∈ (0, 1, ..., m − 1) do              ▷ Eliminate column
21:             m ← A_{r,c} ∧biggerOrEqualsZero(row)
22:             mask ← (m, ..., m)
23:             eq ← isEqual(r, row)

24:             t ← currentRow ⊗ A_r
25:             t ←mux(eq, t, mask ∧ currentRow)   ▷ Don't eliminate pivot
26:             A_r ← A_r ⊗ t
27:         end for
28:         row ← row − found
29:     end for
30: end procedure
```

Constant-Time Column Swaps

Unfortunately Algorithm 4.2 does not necessarily result in a matrix in systematic form. Imagine for example the matrix in Example 4.2.1 had a zero column before the last column. Gauss-Jordan will then preserve that zero column. We therefore have to move this zero column somewhere to the left to get an identity matrix.

Since column swaps are very expensive, due to the way the matrix is represented in cache, we want to avoid them as much as possible. Instead we proceed as follows. We do a constant-time linear scan of each row and store the position of the first set bit, starting from the right. Using these position we can then perform the necessary swaps. We can, however, perform these swaps more efficiently. Note that after transforming the matrix to systematic form, we transpose it to obtain a generator matrix (see Section 3.1.1 step 3g). So we can perform row swaps instead of column swaps by transposing the matrix first! We then perform ct row swaps based on the positions we stored (similarly to the row swaps during Gaussian elimination). While doing this we perform the same swaps on $\mathbf{p}$, $\mathbf{a}$ and $\mathbf{h}$. This is done similar to the naive permutation explained in Section 4.4.1.

4.2.3. Alternative Gauss-Jordan elimination

Doing a full Gauss-Jordan elimination in constant-time is very slow. This comes from the requirement to always run with the worst case performance, i.e. in $\mathcal{O}(m^2 \cdot n)$ operations. Moreover most of these operations involve memory, that does not fully fit the L1-cache. We therefore implemented an additional approach, by Bernstein et al. [13]. We simply restrict the range of key-pairs to the set of keys that are already in systematic form after the Gauss-Jordan elimination as described above, not requiring any column swaps. This allows for several additional speed-ups other than removing column swaps.

Firstly, this assumes that there are no zero columns in our rightmost $m \times m$-submatrix, in essence we assume that the rightmost $m \times m$-submatrix is regular. By this assumption, we do not have to iterate over several columns to look for pivots, and thus we do not have to hide over how many columns we had to iterate. This allows us to only iterate over m columns, reducing the asymptotic complexity from $\mathcal{O}(m^2 \cdot n)$ to $\mathcal{O}(m^3)$.[1] Since in our case m is much smaller than n, this already increases performance a lot.

Secondly, this means that we always do a constant amount of work for every row we look at, and thus we do not have to hide which row we are working on, i.e. we can get rid of the loop in Line 5 of Algorithm 4.2, as well as the **currentRow** variable and the copies necessary in its usage. This reduces the work per column by about a third.

[1] We have ignored the cost of performing the elimination on the remaining $m \times (n-m)$-matrix. However for our choice of parameters and due to vectorization, even in the non-AVX2 version, this is possible in $\mathcal{O}(m^3)$ in total.

Thirdly, it allows us to reduce the loop bounds in Line 10. Originally, we had to scan the whole matrix, otherwise we would have leaked the current row. Since the current row index is now public, we do not have to look at the rows below the current row to hide the row index. Pseudocode for the alternative Gauss-Jordan algorithm is shown in Algorithm 4.3

The problem is sampling those restricted keys. We chose to use the same approach as Classic McEliece, namely we just assume that our assumptions hold during Gauss-Jordan elimination, and abort early and return an error when we notice that they are violated during the elimination, i.e. if we come across an all zero column we immediately return an error. This error is then handled by the key-generation function to trigger a restart of key-generation, resampling all randomness. This way it is ensured that the information about an error cannot leak anything about the actual used randomness, since the randomness being used comes from a run without error with all new randomness.

Note that this approach does not in fact lead to any reduction in the size of the key-space. NTS-KEM already has this restriction, it simply chooses to "fix" the sampled key by changing the random permutation, when it does not lead to such a matrix. It simply turns out that when trying to construct systematic codes in constant-time, the randomized approach is much faster than "fixing" the permutation. This is due to speedups in the Gauss-Jordan elimination.

Algorithm 4.3 Alternative Gauss-Jordan algorithm.

Require: $\mathbf{A}$ is an $m \times n$ matrix, $m < n$.
Ensure: If rightmost $m \times m$ submatrix of $\mathbf{A}$ is regular, $\mathbf{A}$ is systematic.

```
1:  procedure SYSTEMATIC(A)
2:      row = m − 1
3:      for c ∈ (n − 1, n − 2, ..., n − m) do
4:          found ← 0
5:          for r ∈ (row, row − 1, ..., 0) do
6:              m ← A_{r,c}
7:              found ← found ⊗ m
8:              mask ← (m, ..., m)
9:              t ← A_r ⊗ A_row
10:             A_r ← A_r ⊗(t ∧ mask)
11:             A_row ← A_row ⊗(t ∧ mask)
12:         end for
13:         if NOT(found) then
14:             return Error
15:         end if
16:         for r ∈ (0, 1, ..., m) do
17:             m ← A_{r,c} ∧NOTEQUAL(row, r)
18:             mask ← (m, ..., m)
19:             A_r ← A_r ⊗(A_row ∧ mask)
20:         end for
21:         row ← row − 1
22:     end for
23: end procedure
```

4.3. GCD

While computations making use of the GCD are common in cryptography there is surprisingly little in the literature on computing the GCD in constant-time. This might be because the extended Euclidian algorithm is often used to compute the modular inverse, and constant-time implementations then often use different approaches to compute the modular inverse. Moreover, most of the literature related to constant-time GCD computation actually treats variants of the GCD computation that are used to compute the modular inverse. We choose to use a constant-time GCD algorithm by Bernstein and Yang [12], which to our knowledge is the only constant-time algorithm in the literature that actually computes the GCD and not the modular inverse.

For a detailed discussion and a proof of correctness we refer to their paper. We will only give a brief overview of the necessary steps to implement the algorithm.

We begin by defining the function divstep : $\mathbb{Z}\times\mathbb{F}[[x]]\times\mathbb{F}[[x]] \mapsto \mathbb{Z}\times\mathbb{F}[[x]]\times\mathbb{F}[[x]]$,

where $\mathbb{F}[[x]]$ denotes the ring of univariate formal power series over $\mathbb{F}$.

$$\text{divstep}(\delta, F, G) = \begin{cases} \left(1-\delta, G, \dfrac{G(0)F - F(0)G}{x}\right) & \text{if } \delta > 0 \text{ and G(0) } \neq 0, \\ \left(1+\delta, F, \dfrac{F(0)G - G(0)F}{x}\right) & \text{otherwise.} \end{cases} \tag{4.1}$$

As shown by Bernstein and Yang [12, Appendix C], for polynomials F and G with $d_1 = \deg(F)$, $d_2 = \deg(G)$, and $d_1 > d_2$ a sequence of iterative $2(d_1 - d_2)$ divstep applications with input polynomials $x^{d_0}F(1/x)$ and $x^{d_0-1}G(1/x)$ corresponds to the computation of $F \mod G$ as performed during the polynomial GCD algorithm. They then show that this allows to compute $\deg(\gcd(F, G))$ using $2 \cdot \deg(F) - 1$ iterations of divstep on input polynomials $x^{d_0}F(1/x)$ and $x^{d_0-1}G(1/x)$ [12, Theorem 6.2]. This is sufficient for us, since we only compute the GCD to check that two polynomials are relatively prime, i.e. that the degree of the GCD is zero.

We now show how we implemented their algorithm in constant-time. Note that our version is mostly a C-port of the Sage implementation[2] of Bernstein and Yang, with some adjustments to handle our representations of polynomials. In the following we will use pseudocode to abstract away some aspects of C (e.g. memory management, null pointer checks, …), which just make the algorithms less readable. The implementation consists primarily of two functions, a constant-time implementation of divstep, as well as a polynomial reversion (i.e. a function that computes $x^d F(1/x)$). A GCDDEGREE (see Algorithm 4.4) function then simply wraps the calls to these functions.

Algorithm 4.4 gcdDegree function.

Require: Polynomials A, B, with $d = \deg(A) > \deg(B)$.
Ensure: Returns $\gcd(A, B)$.

```
1: function GCDDEGREE(A, B)
2:     F ← REVERSE(A, d)
3:     G ← REVERSE(B, d − 1)
4:     (δ, F, G) ← DIVSTEPSX(2d − 1, 2d − 1, 1, F, G)
5:     return δ >> 1
6: end function
```

We represent a polynomial $\sum_0^d a_i x^i$ as a struct which contains a fixed size array, whose length is an implementation parameter, storing the vector $\mathbf{a} = (a_0, a_1 \ldots, a_d, 0, \ldots, 0)$, and the degree. Computing $x^d A(1/x)$ is then easily implemented by reversing the order of the coefficients and moving the coefficients by $d - \deg(A)$ positions as shown in Algorithm 4.5. Note that this function is

[2] Which is not constant-time due to the use of non-constant-time built-in Sage functions, as pointed out by Bernstein and Yang.

not constant-time with respect to d. This is not a problem for security, though, since we only call it with $d = \tau$ and $d = \tau - 1$, which are public parameters. It is constant-time with respect to the input polynomial. The DIVSTEPSX function in

Algorithm 4.5 Function to compute $x^d A(1/x)$.

Require: Polynomial A with $d \geq \deg(A)$.
Ensure: Returns $x^d A(1/x)$.

```
1:  function REVERSE(A, d)
2:      INITPOLY(F)
3:      for i ∈ (0, 1, ..., d) do                ▷ Reverse and move coefficients
4:          F.coeff[i] ← A.coeff[d − i]
5:      end for
6:      F.deg ← d
7:      zero ← 1
8:      for i ∈ d, d − 1, ..., 0 do              ▷ Update the degree in constant-time
9:          zero ← zero ∧EQUALSZERO(F.coeff[i])
10:         F.deg ← F.deg − zero
11:     end for
12:     return F
13: end function
```

Algorithm 4.6 performs n iterative divstep evaluation, where the returned polynomial has at most degree t. We use a mux-like construction in Lines 9, 10 and 12 to implement the branching from the divstep definition in Eq. (4.1). Note that in Line 15 we do not need to conditionally swap the order of the operands in the subtraction of Eq. (4.1), since we are working on polynomials over a field with characteristic 2 and thus subtraction is the same as addition. Apart from that, the implementation is straight-forward, since our bounds are only depending on public parameters.

Our implementation uses the slower constant-time GCD from [12] which is quadratic in the degree of the polynomials. Bernstein and Yang also provide a subquadratic version, which requires a fast polynomial multiplication (for example based on FFT). Due to time constraints, we did not implement the faster version. In the end the slower version was sufficient in terms of performance (see Chapter 5).

Algorithm 4.6 Constant-time pseudocode for computing iterations of divstep.

Require: Polynomials F, G, $n > 0$.
Ensure: Iteratively compute divstep n times on F and G.

1: **function** DIVSTEPSX(n, t, δ, F, G)
2: INITPOLY(F_2)
3: INITPOLY(G_2)
4: $sz = 0$ ▷ used to store sign(δ) $\wedge (G(0) \neq 0)$
5: **for** $i \in (0, 1, \ldots, n)$ **do**
6: $F_2 \leftarrow F$
7: $G_2 \leftarrow G$
8: $sz \leftarrow$LESSTHANZERO$(-\delta) \wedge$ NOTZERO$(G_2.coeff[0])$
9: $\delta \leftarrow (1 + \delta) - ((\delta << 1) \wedge (-sz))$
10: $F.deg \leftarrow F_2.deg \otimes ((-sz) \wedge (F_2.deg \otimes G_2.deg))$
11: **for** $j \in (0, 1, \ldots, F.coeff.size)$ **do**
12: $F.coeff[j] \leftarrow F_2.coeff[j] \otimes ((-sz) \wedge (F_2.coeff[j] \otimes G_2.coeff[j]))$
13: **end for**
14: **for** $j \in (0, 1, \ldots, G.coeff.size - 1)$ **do**
15: $G.coeff[j] \leftarrow F_2.coeff[0] \cdot G_2.coeff[j+1] + G_2.coeff[0] \cdot F_2.coeff[j+1]$
16: **end for**
17: UPDATEPOLYDEGREE(G, t)
18: **end for**
19: **return** (δ, F, G)
20: **end function**

4.4. Permutations

Performing fast permutations in constant-time is very important, since the security of NTS-KEM relies on the permutation being secret. In this section we want to permute an array or a bit-vector according to a permutation π on the index set. NTS-KEM represents this permutation as a permutation vector $\mathbf{p}$, i.e. an array of size n containing every value in $\mathbb{N}_n$ exactly once. Applying this permutation to an array is then defined as $\pi_{\mathbf{p}}(\mathbf{a}) = (a_{p_0}, a_{p_1}, \ldots, a_{p_{n-1}})$. Unfortunately, doing this in constant-time is not a straight-forward task, since we are accessing memory locations in a secret-dependent way.

4.4.1. Naive Approach

The first way to fix this is ensuring that we are actually accessing the whole array whenever we access an element of the permutation. This can be achieved with the code in Listing 4.4. We utilize our constant-time `mux` function, to ensure that the compiler does not optimize our memory accesses away and conditionally select the correct element to keep in the variable `r`. The obvious issue with this approach is that `permute` in Listing 4.5 needs n^2 memory accesses, which makes it very slow.

4.4.2. Sorting Networks

An alternative to the naive permutation are permutation and sorting networks as shown by Bernstein, Chou, and Schwabe [10].

Definition 4.4.1 (Permutation Network)**.** Let cswap be a gate that takes 3 inputs a, b, c, and produces 2 outputs a', b'. a, b, a', b' are of bit length n and c is 1 bit. cswap swaps the values of a and b if c is set, i.e.

$$\mathsf{cswap}(a, b, c) := \mathsf{if}\, c\, \mathsf{then}\, a' = b; b' = a; \mathsf{else}\, a' = a; b' = b; \mathsf{end}$$

Listing 4.4 – Access an array in constant time

```
1  uint16_t array_access(uint16_t* array, uint32_t pos, size_t len
       ↪ ) {
2    uint32_t b;
3    uint16_t r = 0;
4    for(size_t i = 0; i < len; i++) {
5      b = is_equal(i, pos);
6      r = mux(b, array[i], r);
7    }
8    return r;
9  }
```

Listing 4.5 – Naive Permutation using constant-time array access from Listing 4.4

```
1  void permute(uint16_t* a, uint16_t* a_prime, uint16_t* p,
       ↪ size_t len) {
2    for (size_t i = 0; i < len; i++){
3      a[i] = array_access(a_prime, p[i], len);
4    }
5  }
```

A *permutation network* is a circuit that consists solely of cswap gates and can compute any fixed permutation p by fixing the inputs c to the cswap gates to some values.

Definition 4.4.2 (Sorting Network)**.** Let comp be a gate that takes 2 inputs a, b, and produces 2 outputs a', b', all of bit length n. comp sorts a and b, i.e.

$$\mathsf{comp}(a, b) := \mathsf{if}\ a \leq b\ \mathsf{then}\ a' = a; b' = b; \mathsf{else}\ a' = b; b' = a; \mathsf{end}$$

A *sorting network* is a circuit consisting only of comp gates, that sorts its inputs.

Note that any sorting network can be turned into a permutation network, by exchanging the comp gates with cswap gates. The inputs c can then be computed by applying the original sorting network to the permutation p and storing $c := a > b$ at each comp gates. Furthermore, when implemented in software, with constant-time comp or cswap, sorting and permutation networks are constant-time by design, since the memory access pattern is fixed and only dependent on the number of inputs.

The canonical choice when using a permutation network is the Beneš network [47], due to its optimality. The downside of using such a permutation network is the requirement to transform the permutation vector into the controlbits, which can result in a complicated algorithm if said transformation is to be done in an optimal way. Permutation networks based on sorting networks are usually not optimal, however the transformation from permutation vector to controlbits is easier to implement, since it is simply book-keeping while performing the sorting algorithm. Additionally, the sorting network based approach also allows to permute without the need for precomputation of the controlbits, by simply mirroring the swaps performed in the permutation network onto the array that we want to permute.

When using a sorting network, we have a wide array of options to choose from. Technically, bubblesort is a sorting network, though networks like bitonic sort or odd-even mergesort [8] are obviously preferable due to their asymptotic complexity of only $\mathcal{O}(n \log(n)^2)$. While sorting networks that achieve an asymptotic complexity of $\mathcal{O}(n \log(n))$ are possible, e.g. in the form of the AKS network [1], these are not practically relevant due to large constant factors. As pointed out by Knuth [27, Section 5.3.4, Minimum-comparison networks], odd-even mergesort

performs better in practice and works for arbitrary network sizes (as long as they are a power of 2). It also performs less comparisons than other networks with the same asymptotic complexity [31], which is why we chose odd-even mergesort.

Odd-Even Mergesort

Algorithm 4.7 Recursive odd-even mergesort

Require: $n = 2^k$ for $k \in \mathbb{N} \setminus \{0\}$
Require: The two halves $(a_i)_{0 \leq i < \frac{n}{2}}$ and $(a_i)_{\frac{n}{2} \leq i < n}$, are sorted.
Ensure: The sequence $(a_i)_{0 \leq i < n}$ is sorted.

procedure ODD-EVEN-MERGE($(a_0, a_1, \ldots, a_{n-1}), n$)
  if $n > 2$ **then**
    ODD-EVEN-MERGE($(a_0, a_2, \ldots, a_{n-2}), \frac{n}{2}$)
    ODD-EVEN-MERGE($(a_1, a_3, \ldots, a_{n-1}), \frac{n}{2}$)
    for all $i \in \{1, 3, \ldots, n-3\}$ **do**
      COMP(a_i, a_{i+1})
    end for
  else
    COMP(a_i, a_{i+1})
  end if
end procedure

Require: $n = 2^k$ for $k \in \mathbb{N}$.
Ensure: The sequence $(a_i)_{0 \leq i < n}$ is sorted.

procedure ODD-EVEN-MERGESORT($(a_0, a_1, \ldots, a_{n-1}), n$)
  if $n > 1$ **then**
    ODD-EVEN-MERGESORT($\left(a_0, a_1, \ldots, a_{\frac{n}{2}-1}\right), \frac{n}{2}$)
    ODD-EVEN-MERGESORT($\left(a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \ldots, a_{n-1}\right), \frac{n}{2}$)
    ODD-EVEN-MERGE($(a_0, a_1, \ldots, a_{n-1}), n$)
  end if
end procedure

Our implementation is based on Batcher's odd-even mergesort [8] shown in Algorithm 4.7. Similar to classical mergesort, odd-even mergesort sorts arrays of size n^k for $k \in \mathbb{N}$, by splitting the input array into two halves, recursively sorting those halves, and then merging the sorted halves into a sorted list. Classical mergesort achieves this merge step by linearly scanning the halves, which allows the mergestep to merge two halves of size $\frac{n}{2}$ each in $\mathcal{O}(n)$ operations. This comes with a data-dependent memory access pattern, making it potentially vulnerable to cache-timing attacks. Odd-even mergesort instead merges the two halves by recursively merging the odd and the even subarray using a fixed access pattern (only dependent on n), visualized in Fig. 4.1 for $n = 16$. This fixed access pattern leads to a complexity of $\mathcal{O}(n \log(n))$, which increases the overall complexity from

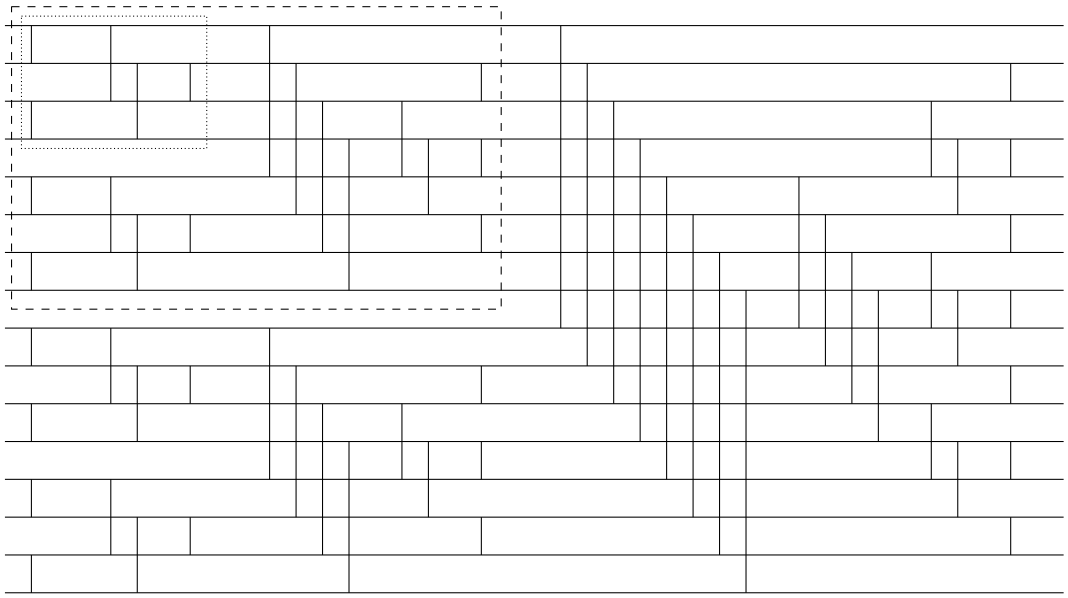


Figure 4.1. – Odd-even mergesort for 16 inputs. Each horizontal line represents one array position, vertical lines represent comp gates. Inputs on the left, outputs on the right side. The dashed box corresponds to odd-even mergesort for 8 inputs, the dotted box to 4 inputs.

mergesort's $\mathcal{O}(n \log(n))$ to $\mathcal{O}(n \log(n)^2)$ for odd-even-mergesort. However, as can be seen from the comparison pattern in Fig. 4.1 many of the comparisons are not dependent on each other and can thus be parallelized, leading to a parallel runtime of $\mathcal{O}(\log(n)^2)$.

A correctness proof of the mergestep is omitted, and can be found in [27] (Section 5.3.4. – Networks for Sorting). The following examples show how odd-even mergesort works.

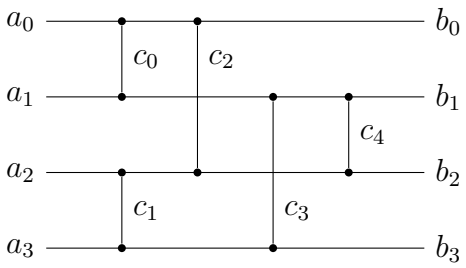


Figure 4.2. – Odd-even mergesort for 4 inputs. Corresponding to the dotted box in Fig. 4.1. Inputs a_i on the left, outputs b_i on the right.

Example 4.4.1 (Odd-Even Mergesort for 4 Inputs)**.** Figure 4.2 shows the odd-even mergesort network for 4 inputs. Let $\mathbf{p} \coloneqq [3, 1, 0, 2]$ and let a'_i, a''_i denote intermediate values on the wires, then applying the network to $\mathbf{p}$, i.e. setting $\mathbf{a} \coloneqq \mathbf{p}$, we get:

$$
\begin{aligned}
&c_0 \coloneqq (a_0 > a_1) = (3 > 1) = 1; && a'_0 \coloneqq 1,\ a'_1 \coloneqq 3\\
&c_1 \coloneqq (a_2 > a_3) = (0 > 2) = 0; && a'_2 \coloneqq 0,\ a'_3 \coloneqq 2\\
&c_2 \coloneqq (a'_0 > a'_2) = (1 > 0) = 1; && a''_0 \coloneqq 0,\ a''_2 \coloneqq 1\\
&c_3 \coloneqq (a'_1 > a'_3) = (3 > 2) = 1; && a''_1 \coloneqq 2,\ a''_3 \coloneqq 3\\
&c_4 \coloneqq (a''_1 > a''_2) = (2 > 1) = 1; && a'''_1 \coloneqq 2,\ a'''_2 \coloneqq 3\\
&b_0 \coloneqq a''_0 = 0,\ b_1 \coloneqq a'''_1 = 1,\ b_2 \coloneqq a'''_2 = 2,\ b_3 \coloneqq a''_3 = 3
\end{aligned}
$$

Thus $\mathbf{b} = [0, 1, 2, 3]$, which is sorted.

If we store the vector $\mathbf{c}\,[c_0, c_1, c_2, c_3]$, we can then use the network to permute by supplying it as input and replacing the comp gates with cswap gates.

Example 4.4.2 (Using Odd-Even Mergesort to Apply a Permutation)**.** Let $\mathbf{a} \coloneqq [a_0, a_1, a_2, a_3]$ with $c \coloneqq [1, 0, 1, 1, 1]$ and $\mathbf{p} \coloneqq [3, 1, 0, 2]$ as in Example 4.4.1, then we get:

$$
\begin{aligned}
&c_0 \coloneqq 1 \longmapsto a'_0 \coloneqq a_1,\ a'_1 \coloneqq a_0\\
&c_1 \coloneqq 0 \longmapsto a'_2 \coloneqq a_2,\ a'_3 \coloneqq a_3\\
&c_2 \coloneqq 1 \longmapsto a''_0 \coloneqq a_2,\ a''_2 \coloneqq a_1\\
&c_3 \coloneqq 1 \longmapsto a''_1 \coloneqq a_3,\ a''_3 \coloneqq a_0\\
&c_4 \coloneqq 1 \longmapsto a'''_1 \coloneqq a_1,\ a'''_2 \coloneqq a_3\\
&b_0 \coloneqq a''_0 = a_2,\ b_1 \coloneqq a'''_1 = a_1,\ b_2 \coloneqq a'''_2 = a_3,\ b_3 \coloneqq a''_3 = a_0
\end{aligned}
$$

And thus $\mathbf{b} = [a_2, a_1, a_3, a_0] = \pi_{\mathbf{p}}^{-1}(\mathbf{a})$, i.e. $b_{p_i} \coloneqq a_i$.

As we can see, when applying a sorting network in this way we actually apply the inverse of the permutation. To see why this happens, think about what we are doing when we compute the controlbits. We are sorting the permutation, i.e.

we are applying a permutation π' on the permutation vector $\mathbf{p}$, so that the result is permutation vector representing the identity. Thus $\pi' = \pi_{\mathbf{p}}^{-1}$, and therefore the steps we record by storing the controlbits are the steps to apply $\pi_{\mathbf{p}}^{-1}$.

Luckily, we can also easily compute $\pi_{\mathbf{p}}(\mathbf{a})$ by traversing the network backwards, i.e. from right to left with inputs b_i and outputs a_i. This works, since we simply undoing the changes that applying $\pi_{\mathbf{p}}^{-1}$ would have done

Example 4.4.3 (Applying the Permutation in Reverse)**.** Let $\mathbf{b} \coloneqq [b_0, b_1, b_2, b_3]$ with $\mathbf{c}$ and $\mathbf{p}$ as in Example 4.4.1.

$$
\begin{aligned}
&c_4 \coloneqq 1 \longmapsto b'_1 \coloneqq b_2,\ b'_2 \coloneqq b_1\\
&c_3 \coloneqq 1 \longmapsto b''_1 \coloneqq b_3,\ b'_3 \coloneqq b_2\\
&c_2 \coloneqq 1 \longmapsto b'_0 \coloneqq b_1,\ b''_2 \coloneqq b_0\\
&c_1 \coloneqq 0 \longmapsto b'''_2 \coloneqq b_0,\ b''_3 \coloneqq b_2\\
&c_0 \coloneqq 1 \longmapsto b''_0 \coloneqq b_3,\ b'''_1 \coloneqq b_1\\
&a_0 \coloneqq b''_0 = b_3,\ a_1 \coloneqq b'''_1 = b_1,\ a_2 \coloneqq b'''_2 = b_0,\ a_3 \coloneqq b''_3 = b_2
\end{aligned}
$$

we get $\mathbf{a} = [b_3, b_1, b_0, b_2] = \pi_{\mathbf{p}}(\mathbf{b})$, i.e. $a_i \coloneqq b_{p_i}$.

Implementation

We use odd-even mergesort both for the permutation of the vectors $\mathbf{a}$ and $\mathbf{h}$ during key generation (issue 5), and for permuting the error-vector $\mathbf{e}$ during decapsulation (issue 13). In both cases we utilize an iterative implementation of odd-even mergesort.

Key-Generation To permute during key-generation we do not precompute the results of the comparisons $\mathbf{c}$. Instead our implementation (see Listing 4.6) simply takes two arrays `val` and `key` as input and sorts `key` (i.e. performs comparisons and swaps on `key`) while mirroring all swaps done on `key` to `val`.

We call the sets of comparisons that do not depend on each other and can be done in parallel *steps*. A set of steps corresponding to one call of Odd-Even-Merge in the body of Odd-Even-Mergesort is called a *stage*. We number the stages logarithmically, meaning stage n is the stage that corresponds to the call Odd-Even-Merge$((a_i)_{0 \le i < 2^n}, 2^n)$. Steps are numbered starting from 0, and relative to their stage, e.g. in Fig. 4.2 c_0 and c_1 are step 0 of stage 1, c_2 and c_3 are step 0 of stage 2, and c_3 is step 1 of stage 2.

The iterative implementation is based on the fact that stage n consists of n steps. Step 0 of stage n compares the i-th element with the $(i + 2^{n-1})$-th element. Step k of stage n compares the i-th element with the $\left(i + 2^{n-k-1}\right)$-the element, excluding the first and last 2^{n-k-1} elements. We can thus simply iterate over the stages with `mergesort` and then iterate over the steps with `merge`. This implementation does not utilize any parallelism. We do however provide an AVX2 version.

Listing 4.6 – Iterative implementation of odd even mergesort without precomputation.

```
1  /**
2   * Permute val by sorting key using odd-even mergesort.
3   * val and key contain 2^n elements.
4  **/
5  void mergesort(uint16_t* val, uint16_t* key, uint32_t n) {
6    for (uint32_t i = 1; i <= n; i++) {
7      for (uint32_t j = 0; j < POW2(n); j += POW2(i)) {
8        merg(val, key, i, j);
9      }
10   }
11 }
12
13 /**
14  * Iterate over the steps of stage n at an offset.
15 **/
16 void merge(uint16_t* val, uint16_t* key, uint32_t n, uint32_t
     ↪ offset) {
17   for (uint32_t i = 0; i < POW2(n-1U); i++){
18     compare(val, key, i, i + POW2(n-1U), offset);
19   }
20   for (uint32_t k = 1; k < n; k++) {
21     for (uint32_t j = 0; j < (POW2(n) - POW2(n-k)); j += POW2(n
         ↪ -k)) {
22       for (uint32_t i = j + POW2(n-k-1U); i < j + POW2(n-k); i
           ↪ ++) {
23         compare(val, key, i, i+POW2(n-k-1U), offset);
24       }
25     }
26   }
27 }
28
29 /**
30  * Compare and swap elements i and j of key at an offset.
      ↪ Mirror the swap to val.
31 **/
32 void compare(uint16_t* val, uint16_t* key, uint32_t i, uint32_t
     ↪  j, uint32_t offset) {
33   uint32_t c = isGreaterThan(key[offset+i], key[offset+j]);
34   uint16_t t_key = key[offset+i] ^ key[offset+j];
35   uint32_t t_val = val[offset+i] ^ val[offset+j];
36   key[offset+i] ^= (-c) & t_key;
37   key[offset+j] ^= (-c) & t_key;
38   val[offset+i] ^= (-c) & t_val;
39   val[offset+j] ^= (-c) & t_val;
40 }
```

When using odd-even mergesort to permute, we are always using it with both 16-bit `key` and 16-bit `val`. Using AVX2's 256 bit-registers we can therefore do up to 16 `compare` operations at once. We omit the code since the only changes to the non-AVX2 version are the types and the fact that Intel's vectorized greaterThan instruction returns -1 instead of 1 in case the comparison is true. We can thus get rid of the negation of `c` in `compare`. The tricky part is loading the correct elements into the AVX2 registers due to odd-even mergesort's somewhat complicated access pattern. For this we distinguish whether we have to do a step 0 or a different step.

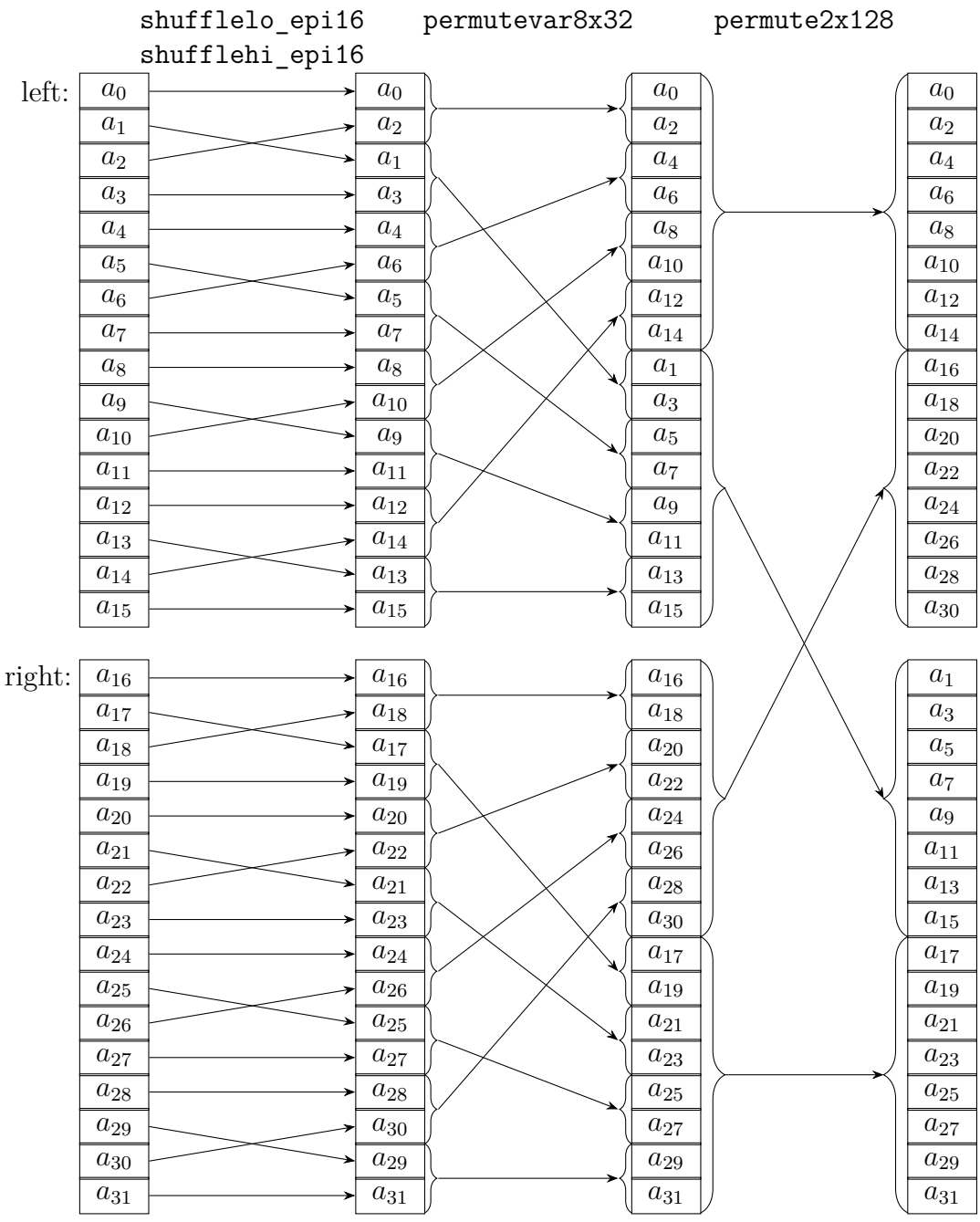
For the zero step our approach loads 16 values (consecutive in memory) per operand into an AVX2 register. Depending on the distance[3] of the elements we need to compare, we now shuffle and permute these two vectors in such a way that we cleanly split the left and right hand side of the comparison into the two registers, so that we can then simply use the vectorized `compare`. E.g. for stage 1, step 0 we want compare adjacent elements ($d = 1$), thus we want load the elements $(a_0, ..., a_{15}), (a_{16}, ..., a_{31})$ into registers and then redistribute them so that we store $(a_0, a_2, ..., a_{30})$ in one register and $(a_1, a_3, ..., a_{31})$ in the other register. We implemented this by using AVX2 `shuffle` and `permute` instruction that interpret the register as 16-bit vectors, 32-bit vectors, and 128-bit vectors to achieve the necessary permutation, as depicted in Fig. 4.3. After applying the conditional swaps, we then apply the inverse of this shuffling to realign the register elements with the corresponding memory elements. Different step sizes between the compared elements can be done analogously, e.g. by skipping the first shuffles in Fig. 4.3 for a distance of 2. Distances of 16 or higher, skip the shuffling altogether.

For the the other steps we are only handling the cases were the distances are 16 or higher, by unrolling the loops by 16 and using the vectorized `compare`. The cases with smaller distances fall back to the sequential version. Since we are working on sorting networks with 4096 or 8192 inputs, these sequential parts make up for only a very small part of the network.

Decapsulation For decapsulation we have to permute bit vectors instead of integer vectors. In principle this can be achieved by simply changing the memory access to `val` in `compare` in Listing 4.6 to only access single bits. Such an implementation is grossly underutilizing the available parallelism, though. Instead our approach utilizes precomputation of the bits controlling the conditional swaps as explained above. Given these bits we can then perform up to 64 conditional swaps in one go, by packing both the control bits and the data bits into 64-bit integers and using bitwise operations as we already do in `compare`. A version implementing this for a network with 64 inputs is shown in Listing 4.7. `left64` and `right64` are bit masks encoding the access pattern, i.e. the i-th bit of `left64[j][k]` is set if a_i is the left argument of a COMP call in step k of stage j (analogously for `right64`). `left64` is shown in Listing 4.8, `right64` is can be computed as

[3] For two elements a_i and a_j, the distance is $d(a_i, a_j) := \|i - j\|$.

Figure 4.3. – Shuffling process to split comparison operands into two registers.



Listing 4.7 – Bitwise odd-even mergesort using precomputed control bits for 64 inputs.

```
1  /**
2   *  @param[in,out]  arr          bit vector to be permuted
3   *  @param[out]     control      controlbits defining the
       ↪ permutation
4  **/
5  void bitPermute(uint64_t arr, const uint64_t* control) {
6    uint64_t l, r, shift, cshift, t, stage, step;
7      for (stage = 0; stage < 6; stage++){
8        for (step = 0; step <= stage; step++) {
9          shift = 1 << (stage - step);
10         // Select the bits to be conditionally swapped.
11         l = arr & left64[stage][step];
12         r = arr & right64[stage][step];
13
14         // Unset selected bits in arr.
15         arr ^= l|r;
16
17         // Conditional swap
18         t = (r >> shift) ^ l;
19         l ^= control[stage][step] & t;
20         r ^= (control[stage][step] & t) << shift;
21
22         // Store the bits in arr
23         arr |= l|r;
24     }
25   }
26 }
```

`left64<<shift`. For ease of implementation we store the control bits in the same pattern as `left64`, i.e. only positions of `control` where `left64` is set are used, the remaining positions are zero. The actual versions used for 4096 and 8192 inputs work in the same principle, though using highly unrolled loops for the remaining cases, as well as storing the bit mask for stages 7 and up in a more compressed format, i.e. not repeating redundant integers.

For computing the control bits we simply use a modified version of `mergesort` from Listing 4.6 that only works on `key` and stores the result in the pattern defined via `left64`. In the AVX2 version only the computation of the bits is vectorized, based on our AVX2 version of odd-even mergesort. Applying the permutation uses the non-AVX2 implementation, since performance was good enough without vectorization.

Listing 4.8 – Bit mask used in Listing 4.7.

```
uint64_t left64[6][6] = {
  {
    0x5555555555555555, 0,
    0, 0,
    0, 0
  },
  {
    0x3333333333333333, 0x2222222222222222,
    0, 0,
    0, 0
  },
  {
    0x0F0F0F0F0F0F0F0F, 0x0C0C0C0C0C0C0C0C,
    0x2A2A2A2A2A2A2A2A, 0,
    0, 0
  },
  {
    0x00FF00FF00FF00FF, 0x00F000F000F000F0,
    0x0CCC0CCC0CCC0CCC, 0x2AAA2AAA2AAA2AAA,
    0, 0
  },
  {
    0x0000FFFF0000FFFF, 0x0000FF000000FF00,
    0x00F0F0F000F0F0F0, 0x0CCCCCCC0CCCCCCC,
    0x2AAAAAAA2AAAAAAA, 0
  },
  {
    0x00000000FFFFFFFF, 0x00000000FFFF0000,
    0x0000FF00FF00FF00, 0x00F0F0F0F0F0F0F0,
    0x0CCCCCCCCCCCCCCC, 0x2AAAAAAAAAAAAAAA
  },
};
```

4.5. Random Sampling

This section discusses the creation of the random permutation π as well as the random error vector $\mathbf{e}$ in a constant-time fashion. For this we assume we have a cryptographically secure source of randomness. In practice this is provided via a pseudo-random number generator (PRNG) which returns random bytes. Note that this allows us to sample an arbitrary number of bits – since we can sample several times and throw away unneeded bits, but means we can only sample in constant time from $\{0,1\}^n$ for arbitrary n. This is due to the well known fact that in general we cannot transform a uniform distribution over an arbitrary finite set into a uniform distribution over a different arbitrary finite set, without using a probabilistic-time algorithm.

Fortunately, the constant-time requirement is stronger than the property we require. Instead we require that an attacker cannot learn any information about the secret (i.e. the randomness in our case) by observing the runtime of our algorithm. This can be formally captured as follows:

Definition 4.5.1. For any (probabilistic) algorithm $\mathcal{A}$, let $T_\mathcal{A}$ be the random variable denoting the runtime and S be the random variable denoting the secrets used by the algorithm. We call $\mathcal{A}$ secure against timing attacks if and only if

$$\mathrm{I}\left(T_\mathcal{A}; S\right) = 0,$$

where $I(\cdot;\cdot)$ denotes the mutual information. Equivalently, we require that $T_\mathcal{A}$ and S are statistically independent (i.e. $\Pr[S \mid T_\mathcal{A}] = \Pr[S]$).

When using this definition care has to be to not overly inflate the secrets S. Including, for example, a length in S, might often not allow proving that the mutual information is zero. (Though in the case of lengths this is consistent with classical security definitions, which often assume the length to be fixed.) It is easy to see that an algorithm that is constant-time fulfills this condition since its runtime is in a sense a fixed public parameter. It is equally easy to come up examples that are not constant-time, but fulfill this definition, showing that it is indeed weaker than constant-timeness. E.g. assume we can sample random bits in constant-time. Then sample two bits independently from each other a and b, where b is the secret. If a is set wait for a fixed amount of time and return b, otherwise return b without waiting. The runtime of this (somewhat nonsensical) example is obviously not constant-time, but it is statistically independent from the secret b since a and b are independent.

While we are not aware that this intuitive definition has been stated in the literature or is used to analyze algorithms, related concepts exist in the form of *Mutual Information Analysis* (MIA) [20]. MIA is a technique used in side-channel analysis which uses measurements of mutual information to extract secrets through a side-channel. Gierlichs et al. [20] also mention that ideally this mutual information would in theory be zero when no leakage occurs.

In practice, showing that an algorithm is constant-time is often much easier than the above property. For this reason constant-timeness is usually required instead. In this section we cannot achieve this, however, and thus aim for the property in Definition 4.5.1.

In the following we will present our solutions to issues 3 and 8. As we will see, these solutions do not depend on the ability to sample uniformly from $\mathbb{N}_k$ for arbitrary k. As such, we do not require the Knuth-Yao algorithm, which solves issue 4. Nonetheless, the Knuth-Yao algorithm is safe to use, since it satisfies Definition 4.5.1. A proof of this can be found in Appendix B.

4.5.1. Random Permutation

First we want to generate a uniformly random permutation over the set of $n = 2^m$ elements (where n and m are NTS-KEM parameters), which we will represent as a permutation vector. An easy way to securely create a random permutation is making the memory accesses in the Fisher-Yates shuffle constant-time. This results in a significant performance degradation though. A faster alternative works by realizing that any finite, non-repeating sequence $\mathbf{s}$ with $s_i \in \mathbb{N}$ implicitly defines a permutation f. We can see this by mapping each element s_i of a sequence $\mathbf{s}$ with $|\mathbf{s}| = n$ to an element of $\mathbb{N}_n$ as follows.

Let $\hat{\mathbf{s}} = (\hat{s}_i)_{i \in \mathbb{N}_n}$ be the ordered sequence of elements of $\mathbf{s}$, i.e. the following conditions hold

1. $\{s | s \in \hat{\mathbf{s}}\} = \{s | s \in \mathbf{s}\}$,
2. $\forall i, j \in \mathbb{N}_n : i < j \longrightarrow \hat{s}_i < \hat{s}_j$.

Then f maps each element of $\mathbf{s}$ to its position in $\hat{\mathbf{s}}$, i.e. $f(s) \coloneqq i,\ \text{s.t. } \hat{s}_i = s$.

Example 4.5.1. Let $\mathbf{s} \coloneqq (8, 3, 4, 1)$, then $\hat{\mathbf{s}} = (1, 3, 4, 8)$. And thus we map $\mathbf{s}$ to f represented by the permutation vector $\mathbf{f} = (f(s_1), ..., f(s_n)) = (3, 1, 2, 0)$, .

Therefore we can simply repeatedly sample n integers until all n integers are distinct to create $\mathbf{s}$, and then sort $\hat{\mathbf{s}}$ to define f (represented by a vector $\mathbf{f}$). To decrease the probability of collisions, we sample 64-bit integers. Using the fact that $n = 2^m$ and the birthday bound, we can bound the probability p_n of collision (and thus of resampling) by

$$p_n \leq \frac{n^2}{2^{65}} = \frac{2^{2m}}{2^{65}} = \begin{cases} 2^{-41} & \text{if } m = 12 \\ 2^{-39} & \text{if } m = 13 \end{cases}.$$

To compute the mapping from random integers to a permutation, we simply use odd-even mergesort (see Section 4.4.2) to sort $\mathbf{s}$ and mirror the swaps to $\mathbf{p} = (0, 1, ..., n-1)$ in constant time. Since this way we also sort $\mathbf{s}$, we only check for duplicates in $\mathbf{s}$ after sorting, which is then done using a simple linear scan. The whole process is depicted in Listing 4.9.

Listing 4.9 – Creating a uniformly random permutation.

```
// N is the NTS-KEM parameter n
// M is the NTS-KEM parameter m, i.e. m = log(n)
void create_permutation(uint16_t* p) {
  uint64_t ran[N];
  uint64_t t;
  do{
    t = 1;
    randombytes((uint8_t*) ran, sizeof(ran));
    for(int i = 0; i < N; i++) {
      p[i] = i;
    }
    mergesort(p, ran, M);
    for(int i = 1; i < N; i++) {
      t &= isNotEqual(ran[i], ran[i-1]);
    }
  } while (not(t));
}
```

4.5.2. Random Error Vector

Here the problem is to generate a uniformly random bit vector $\mathbf{e}$ of length n with Hamming weight τ. Using a constant-time version of the Fisher-Yates shuffle, which ensures to access every array element for every swap, was too slow. Instead, if we can sample the τ distinct indices where the vector will be set, we can then simply set these bits. Setting the bits is easy to do in constant time, by simply checking for every index whether it is in the list of sampled indices. This check is done using a simple linear scan. We then only have to continue with loop when an index is found instead of stopping early, as depicted below.

```
// N = NTS-KEM paramter n/8
// char e[N] will store the error as bit-vector
// uint64_t idx[tau] contains the sampled indices
for (int i = 0; i < N; i++) {
  e[i] = 0;
  for (int j = 0; j < tau; j++) {
    c = -isEqual(i, idx[j] >> 3);
    e[i] |= (1 << (idx[j] & 7)) & c;
  }
}
```

This problem then boils down to sampling τ distinct indices. For this we use the approach used in the Classic McEliece implementation [13], which works as follows.

1. Sample 2τ random numbers from $\{0, ...n\}$.

2. Check if there are at least τ distinct numbers, this is done by checking for every element whether it is equal to any of the elements before it.

3. If we have do not have enough elements go to step 1, otherwise pick the first τ distinct elements.

Note that this approach fulfills Definition 4.5.1, since one iteration of this sampling process is constant-time and the randomness from all but the last iteration is not used.

We chose the parameter 2τ in step 1 the same way as the Classic McEliece team did. Simply sampling τ would lead to a high probability of retries in step 3 as that case would reduce to the birthday problem, for which we can bound the probability of collision p_{coll} by $1 - \mathbf{e}^{-\tau\cdot(\tau-1)/2n} \leq p_{coll} \leq \frac{\tau^2}{2n}$. For $\tau = 136$ and $n = 8192$, which has the highest bounds, this results in bounds of 0.67 and 1.13. By instead sampling 2τ numbers the probability of at least τ distinct numbers is empirically very close to 1. And thus the probability of retrying in step 3 is close to 0. The additional cost of sampling twice as many numbers should be negligible.

To sample random numbers in step 1 we tried several approaches. First we used simple rejection sampling. This was significantly slower than the Knuth-Yao algorithm. Note that as stated above and shown in Appendix B the Knuth-Yao algorithm is actually safe to use here. We found an even simpler and faster approach, though. We can use the fact that, in NTS-KEM, n is always a power of two (2^{12} or 2^{13}). This means we can simply sample 2 byte integers and zero the additional upper bits.

4.6. Verification

We used ctgrind to successfully verify our implementation on optimization levels `-O0` to `-O3` when compiled with gcc-6.3. As part of our implementation, we also provide scripts to automatically run ctgrind with the appropriate settings. As discussed in Section 2.3, memcheck can produce false positives when using vector registers. On higher optimization levels the compiler tries to vectorize loops on its own, which can trigger these false positives. In our case ctgrind reported issues for the 13-80 and 13-136 NTS-KEM version's bitsliced Berlekamp-Massey algorithm. We manually inspected the produced assembly to ensure that these were actually false positives. Similarly ctgrind reports issues in the avx2 version which are also false positives.

We can only give guarantees for the compiler and optimizations we tested (i.e. gcc-6.3 with the above optimizations). When using different compilers we recommend running ctgrind using the provided scripts to verify that no issues are introduced by the compiler. In case of issues we recommend inspecting the assembly to ensure that the issues are false positives due to vectorization. For convenience we also provide ways to deactivate loop vectorization during compilation. We provide an optional make target `use_pragma` which uses pragmas to instruct

the compiler to not vectorize the specific sections of the code where false positives were caused in our case. These pragmas are implemented to work with gcc and clang. Additionally the makefiles provide targets `no_loop_vectorize_gcc` and `no_loop_vectorize_clang`, which fully deactivate loop vectorization during compilation. Note however that both ways of deactivating loop vectorization can reduce performance (see Section 5.3.5) and inspecting the assembly to ensure that any reports generated by ctgrind are false positives is preferable.

5. Performance Evaluation

5.1. Methodology

We instrumented our implementation at several points in the program to collect runtime data. Namely we collect average and total runtime of code sections, as well as minimum, maximum value observed. Additionally we compute the median and the 1st and 3rd quartiles of the observed runtimes. These are computed on the fly using the P^2 algorithm by Jain and Chlamtac [24]. Our machine is an Intel Core i5-4570 CPU @ 3.20GHz with 16 GB RAM.

The evaluation setup is as follows. For one configuration of our implementation a single run consists of performing key-generation, encapsulation with that key, and decapsulation of the encapsulated ciphertext with the corresponding key. We use 1000 such runs per configuration. The selection of configurations is depicted in Table 5.1

Table 5.1. – Configurations used for the performance evaluation.

non-ct	Optimized and AVX2 implementations of NTS-KEM.
ct	Constant-time version as explained in Chapter 4. Uses the rejection sampling approach for Gaussian elimination.
rref	like ct, but using a full Gaussian elimination, with column swaps at the end.
reduced-randomness	like ct, but using 32-bit random integers during the generation of the random permutation.
Fisher-Yates	like ct, but using a constant-time Fisher-Yates shuffle to create the random permutation.
no-sorting-network	like ct, but using the naive constant-time permutation instead of sorting networks to permute **a** during key-generation.
Knuth-Yao	like ct, but using Knuth-Yao to sample bounded integers when creating the error vector.
rejection-sampling	like ct, but using rejection sampling to sample bounded integers when creating the error vector.
no-bit-sorting-net	like ct, but using a naive ct bit-vector permutation instead of sorting networks during decapsulation.
bit-shuffle	like ct, but using the constant-time Fisher-Yates shuffle to create the random vector.
pragma	like shuffle, but using pragmas to disable loop vectorization locally to remove false positives.
no-loop-vectorize	like shuffle, but deactivating loop vectorization globally.

Table 5.2. – Overview over mean number of cycles measured for non constant-time NTS-KEM as per reference implementation.

		Key-Generation	Encapsulation	Decapsulation
12-64	optimized	87 119 976	138 052	1 129 596
	avx2	76 377 368	176 495	967 353
13-80	optimized	257 709 964	552 701	2 862 617
	avx2	225 757 666	641 278	2 490 900
13-136	optimized	437 865 095	934 001	5 513 895
	avx2	371 496 148	976 928	4 236 917

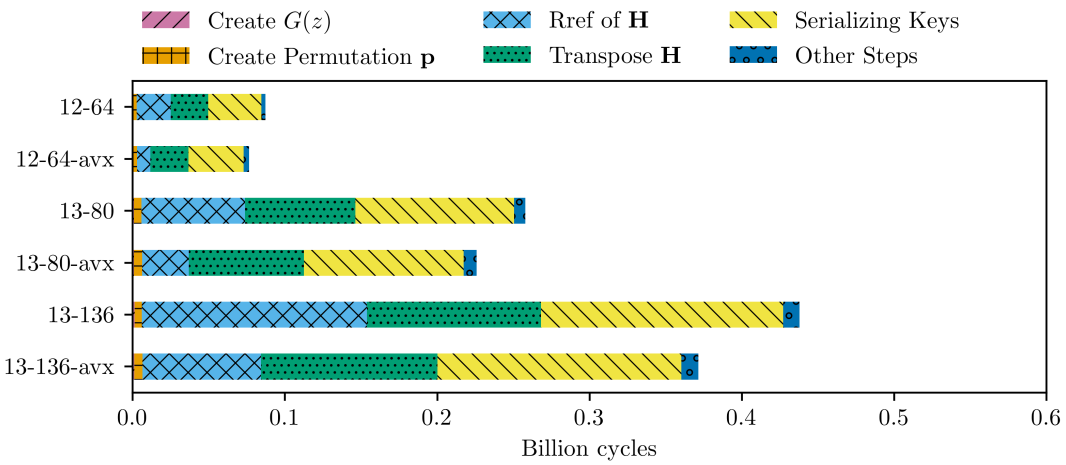


Figure 5.1. – Mean time spent in key-generation for non-ct NTS-KEM.

5.2. Baseline – Non Constant-Time NTS-KEM

We start with presenting the performance of the original NTS-KEM implementation on our machine, which we use as a baseline. Table 5.2 gives an overview of the total timings. Note that our timings are slightly higher than those reported in the submission [4], which is most likely due to our instrumentation being more involved. In the following we will briefly go over the bottlenecks of NTS-KEM.

Key-Generation

Figure 5.1 shows the mean timings for the important steps of key-generation. As we can see the bottleneck lies in creating the matrix, which takes about half of the time needed for key-generation. Again half of that is taken by the M4RI algorithm and the other half by transposing the matrix. Apart from serialization of the keys at the end, the remaining steps do not make up a significant portion

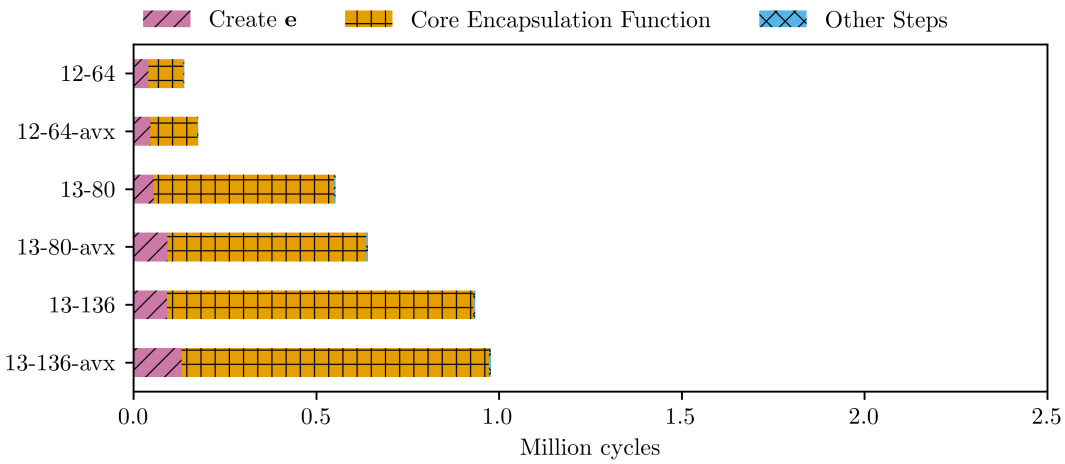


Figure 5.2. – Mean time spent in encapsulation for non-ct NTS-KEM.

of the runtime.

Encapsulation

For encapsulation most of the work is done in the core encapsulation function (see Fig. 5.2). Figure 5.3 shows the timings for the steps of the core encapsulation function. Interestingly the most relevant part of encapsulation is the setup step, which loads the serialized matrix from the public key into a matrix. In addition, the AVX2 version is slightly slower, which is most likely caused by the fact that the AVX2 matrix is larger, since it consists of memory chunks that are a multiple of 256 bit instead of 64 bit. This leads to additional memory being written (i.e. set to zero) during initialization.

Decapsulation

Figure 5.4 gives an overview of the decapsulation timings. Immediately, we can see that syndrome computation makes up about half of the runtime. Other steps contains mostly the setup phase (i.e. reading the key data into data structure) and reencapsulation. The remaining steps only make up a small part of the runtime.

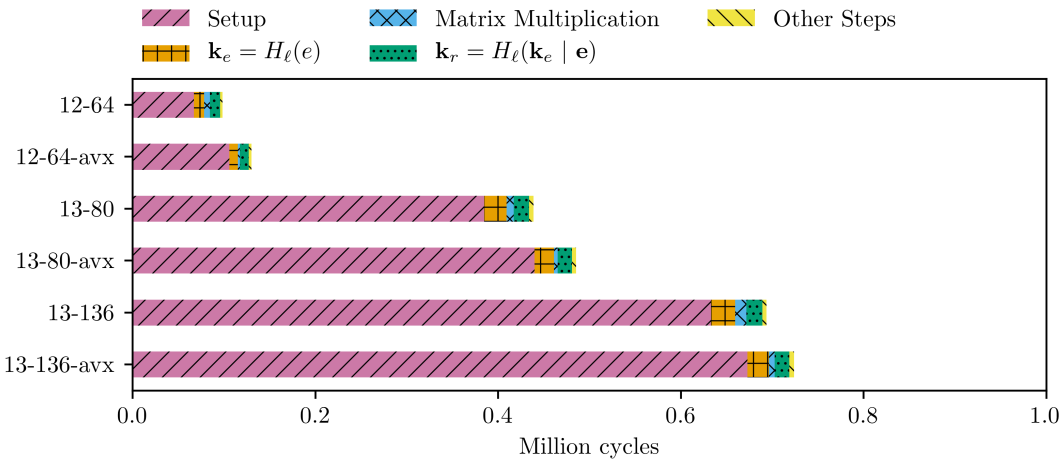


Figure 5.3. – Mean time spent in the core encapsulation function for non-ct NTS-KEM.

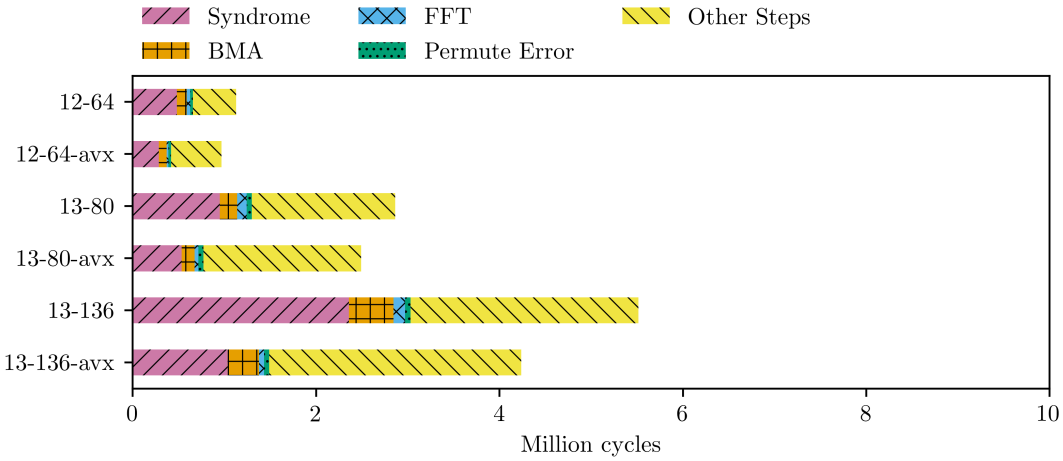


Figure 5.4. – Mean time spent in decapsulation for non-ct NTS-KEM.

5.3. Results – Constant-Time NTS-KEM

We now present the performance of our constant-time version (ct in Table 5.1). We will then follow-up with a discussion of different approaches explained in Chapter 4. A summary of the timings is given in Table 5.3.

Table 5.3. – Overview over mean timings measured for constant-time NTS-KEM.

		Key-Generation	Encapsulation	Decapsulation	Precompute
12-64	opt.	735 926 942	343 526	909 871	1 821 409
	avx2	347 025 387	233 326	697 696	1 445 779
13-80	opt.	2 488 549 580	1 093 456	2 088 743	4 599 707
	avx2	1 227 321 932	933 114	1 649 790	3 599 908
13-136	opt.	6 282 556 806	1 789 491	4 057 654	5 383 955
	avx2	2 806 302 970	1 323 330	2 684 412	4 350 258

Immediately we notice the huge impact of increasing the parallelism with AVX2 especially for key-generation, though key-generation is still immensely slower than in the non-constant-time version, with a factor of up to 10 for the non AVX2 and a factor of about 5-7 for the AVX2 version. Encapsulation times roughly double, while decapsulation time roughly doubles when we include the cost of precomputation.

5.3.1. Key-Generation

Figure 5.5 gives an overview over the time spent during different steps of key-generation. When comparing with the non-constant-time version (Fig. 5.1, note the change of the x-axis scale), we can see that the transformation into systematic form now completely dominates the runtime. This is to be expected given that the extremely fast but unfortunately non-constant-time M4RI algorithm used in the non-constant-time versions already made up a big portion of the runtime. Additionally we can see that the creation of the Goppa polynomial $G(z)$ in constant-time now makes up a noticeable portion of the runtime, where before it was completely insignificant. This is due to our choice of using the simple version of Bernstein and Yang's ct-GCD algorithm. We expect that this could be significantly improved upon by implementing a constant-time polynomial multiplication using FFT and then following Bernstein and Yang's approach for a faster ct-GCD. Performance gains for key-generation as a whole through this would be minimal however, since the rref transformation require most of the time.

Figure 5.6 shows the quartiles of constant-time key-generation. Most importantly we can see that the median runtime is significantly lower than the average runtime, with 50 % of key-generations taking less than 240 000 000, 930 000 000

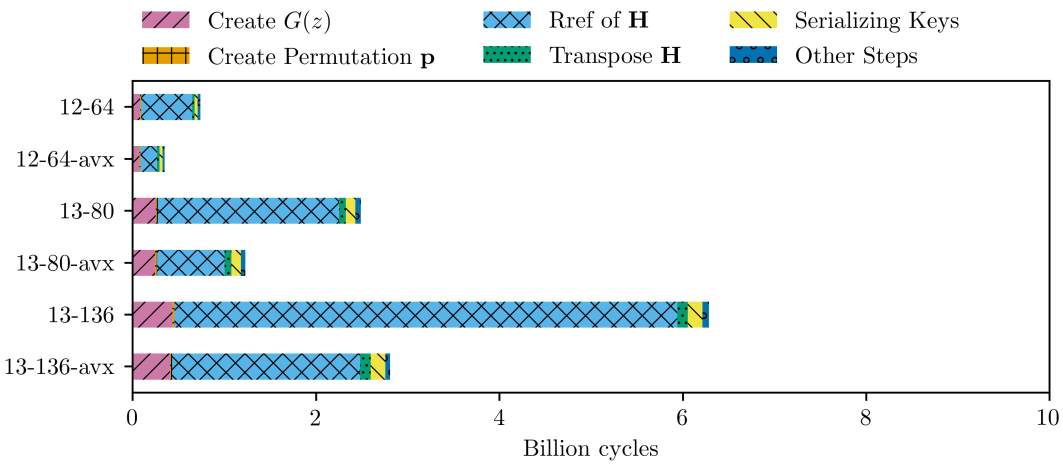


Figure 5.5. – Mean time spent in key-generation for ct NTS-KEM.

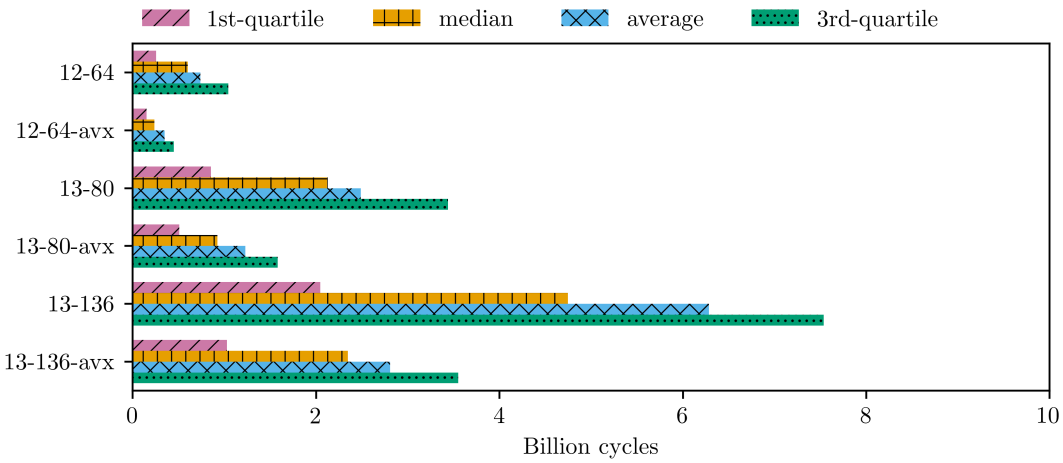


Figure 5.6. – Quartiles of ct key-generation.

and 2 300 000 000 cycles for the three AVX2 versions. This is to be expected given that such a rejection sampling approach should result in a geometric distribution.

On average we needed 3.4 attempts to generate a valid key. This is consistent with the numbers reported by Bernstein et al. [13], who report that about 29 % of randomly chosen matrices can be transformed into systematic form without column swaps.

Gaussian Elimination

We will now briefly compare our two approaches to Gaussian elimination, i.e. the full Gaussian elimination with column swaps and the rejection sampling approach without column swaps. Figure 5.7 shows the timings for the full Gaussian elimina-

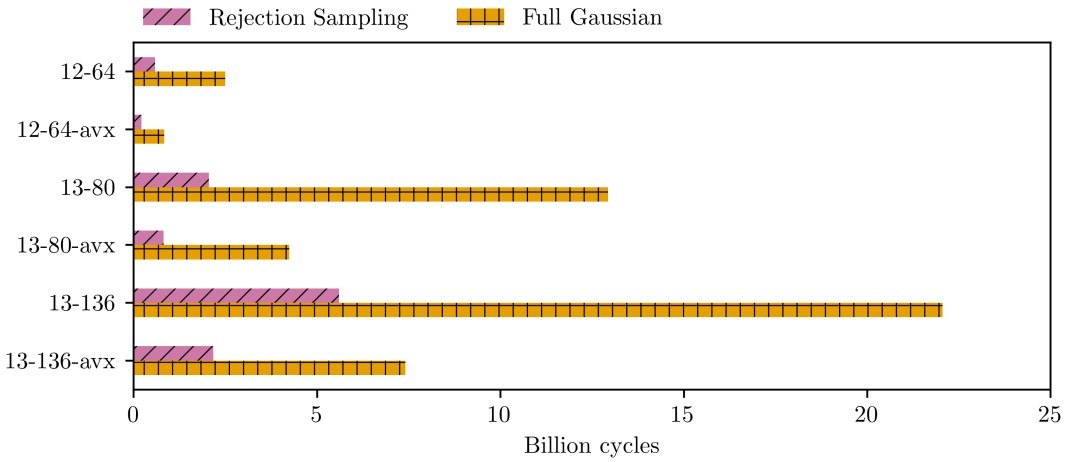


Figure 5.7. – Average time spent transforming a matrix into systematic form in constant-time.

tion contrasted with the rejection sampling approach. As we would expect mean key generation is much slower than in the rejection sampling approach. Since we are working on an $m\tau \times (n - m\tau)$-matrix we would expect a factor roughly corresponding to $\frac{n-m\tau}{m\tau}$ between the two versions, which directly stems from using an $\mathcal{O}((m\tau)^3)$ algorithm instead of an $\mathcal{O}((m\tau)^2(n - m\tau))$ algorithm. This would then come down to factors of 4.3, 6.9 and 3.6 which fits well with our measured factors of 4.3, 6.3 and 3.9 for the non-AVX2 and 4.0, 5.2 and 3.4 for the AVX2 versions.

Creating Random Permutations

Now, we will focus on creation of the random permutation. We will first look at the constant-time Fisher-Yates shuffle in contrast to the sorting network based approach. The time needed for the Fisher-Yates shuffle is depicted in Fig. 5.8. We can immediately see there are huge differences between these two approaches.

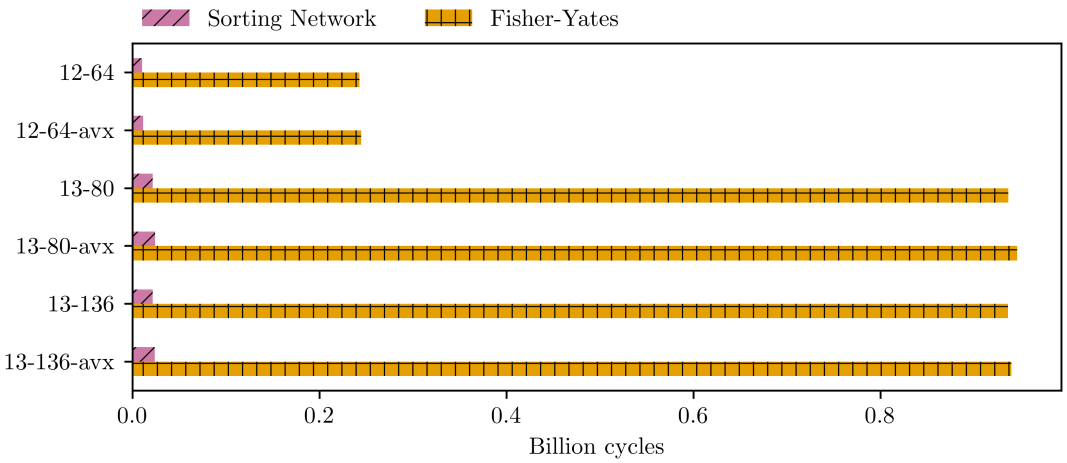


Figure 5.8. – Average time spent generating the random permutation during key-generation for different approaches.

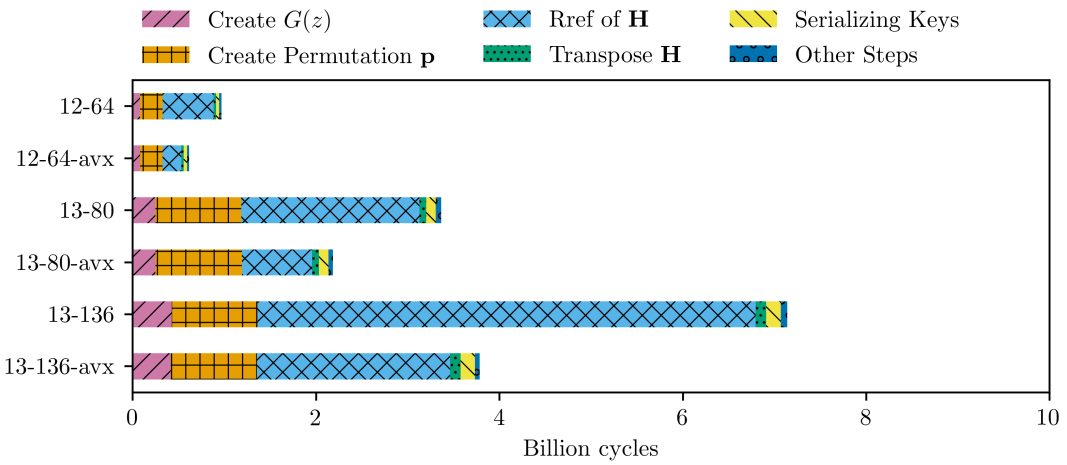


Figure 5.9. – Mean time spent in key-generation for Fisher-Yates configuration of ct NTS-KEM.

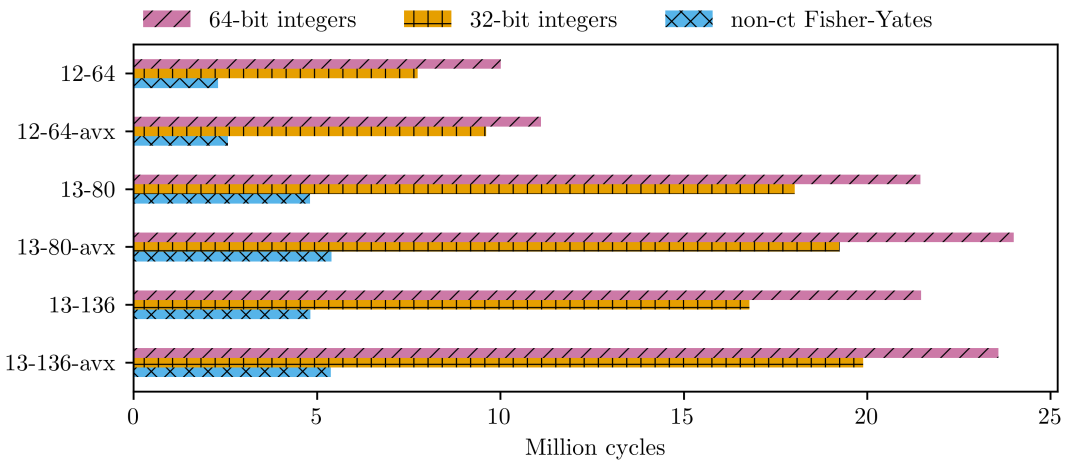


Figure 5.10. – Average time spent generating the random permutation during key-generation using different integer sizes.

This is simply caused by the fact that our constant-time Fisher-Yates accesses the whole array for every swap, which results in a quadratic runtime. Our sorting network based approach only needs $n \log(n)^2$ operations. Since we are talking about memory accesses, these differences are especially relevant. In fact constant-time Fisher-Yates is so bad, that creation of the permutation actually becomes a significant part of the overall key-generation time, as depicted in Fig. 5.9.

Secondly, we compare the performance impact of using 32-bit integers instead of 64-bit integers when creating the random permutation as explained in Section 4.5.1. Remember, that this is done by sampling n integers and sorting them. We expect that the 32-bit version would be slightly faster, since we need to sample half as much randomness and more importantly the sorting network has to access half as much memory. As can be seen in Fig. 5.10 this is true. The difference is small, though, especially when looking at the context of key-generation, where the time needed by creation of the permutation is completely negligible as seen in Fig. 5.5. However, using 32-bit integers our collision probability rises to $\frac{2^{2m}}{2^{33}} = \begin{cases} 2^{-9} & \text{if } m = 12 \\ 2^{-7} & \text{if } m = 13 \end{cases}$ from $\frac{2^{2m}}{2^{65}} = \begin{cases} 2^{-41} & \text{if } m = 12 \\ 2^{-39} & \text{if } m = 13 \end{cases}$ in the 64-bit case. This does not make the 32-bit version slower than the 64-bit version, though. Additionally we have double the memory usage in the 64-bit variant, i.e. 32 and 64 KB instead of 16 and 32 KB (for $m = 12$ and 13 respectively). Given the size of the keys, namely between 321 and 1406 KB depending on the parameter set as reported in the original submission, this is also only a very small increase.

Independent of the choice of integer size our constant-time approach is however significantly slower than the non-constant-time Fisher-Yates shuffle. This is expected, since one try of our approach runs in $\mathcal{O}(n \log(n)^2)$, whereas the Fisher-Yates shuffle runs in $\mathcal{O}(n)$. Still, as a result, our approach using 64-bit integers

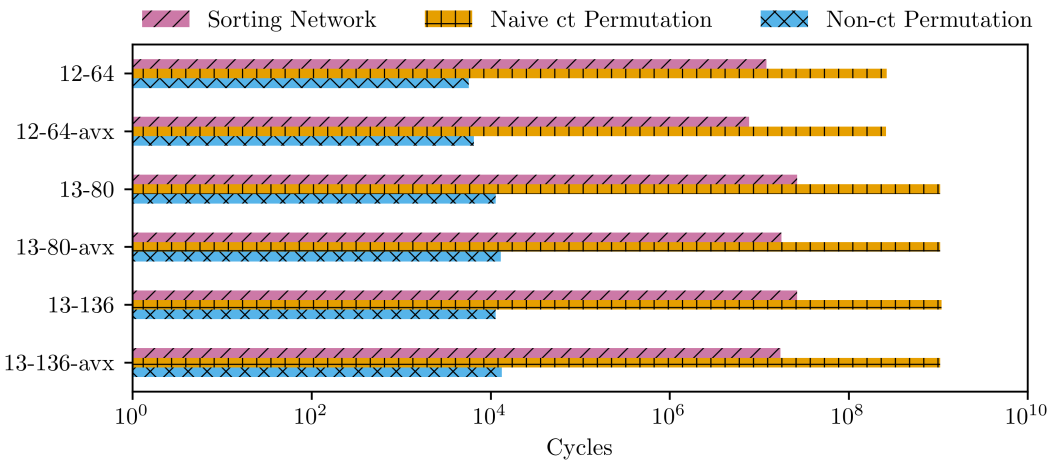


Figure 5.11. – Mean time spent permuting **a**.

is good. Optimizations in this part would not lead to any significant speedups of key-generation as a whole. On very memory constrained devices using the 12-64 version, it might be preferable to use 32-bit integers, however.

Overall, we can conclude that our method of permutation creation performs well. Even with 64-bit integers its runtime is negligible with respect to the total key-generation time, which is bottlenecked by the matrix transformation.

Permuting Elements of $\mathbb{F}_{2^m}$

We will now show the difference in performance between the naive approach to permuting in constant-time and the sorting network based approach. Remember that the naive approach runs in quadratic time similarly to the Fisher-Yates shuffle above. Thus we expect a similarly big difference in performance compared to the sorting network based approach. Figure 5.11 confirms this expectation, with the naive approach being an order of magnitude slower (note the logarithmic scale!). Similarly to the case of ct-Fisher-Yates above, this approach would make permutation of **a** take a significant part of total key-generation time.

Figure 5.11 also shows the difference between the sorting network based constant-time permutation and the non-constant time permutation. We see that unfortunately constant-time permutation is 3 orders of magnitude slower than non-constant-time permutation. However comparing with Fig. 5.5 we see that permuting (as part of 'other steps') makes up for a very small amount of the total key-generation time. Thus in the context of much more expensive Gaussian elimination, our approach to permutation performs reasonably well.

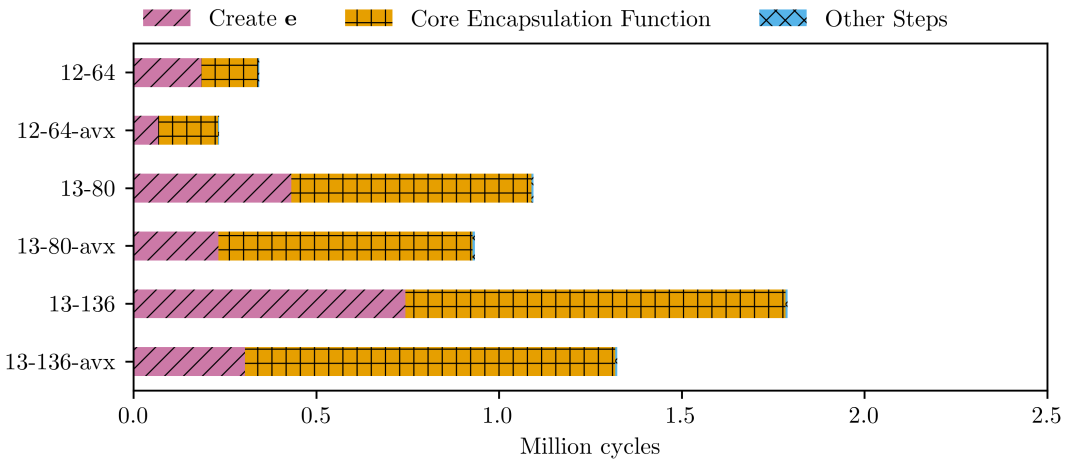


Figure 5.12. – Mean time spent in encapsulation for ct NTS-KEM.

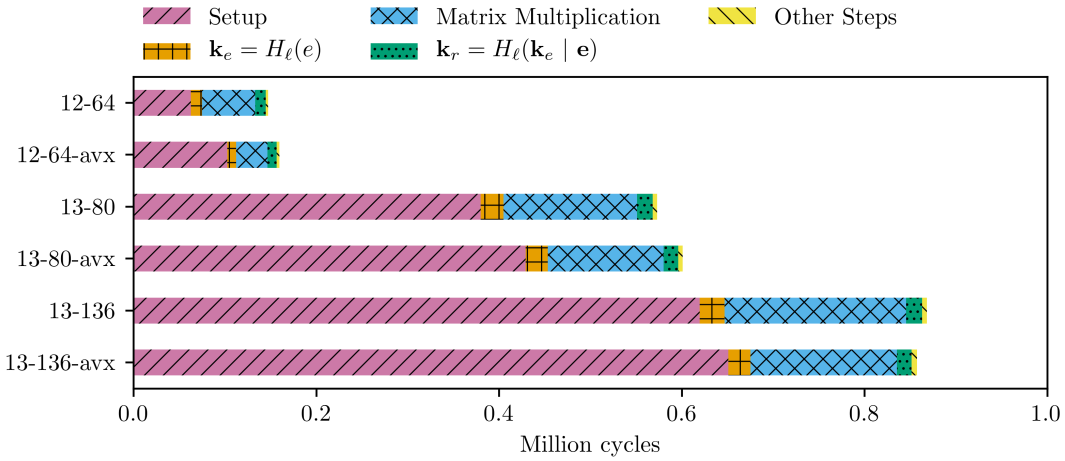


Figure 5.13. – Mean time spent in the core encapsulation function for ct NTS-KEM.

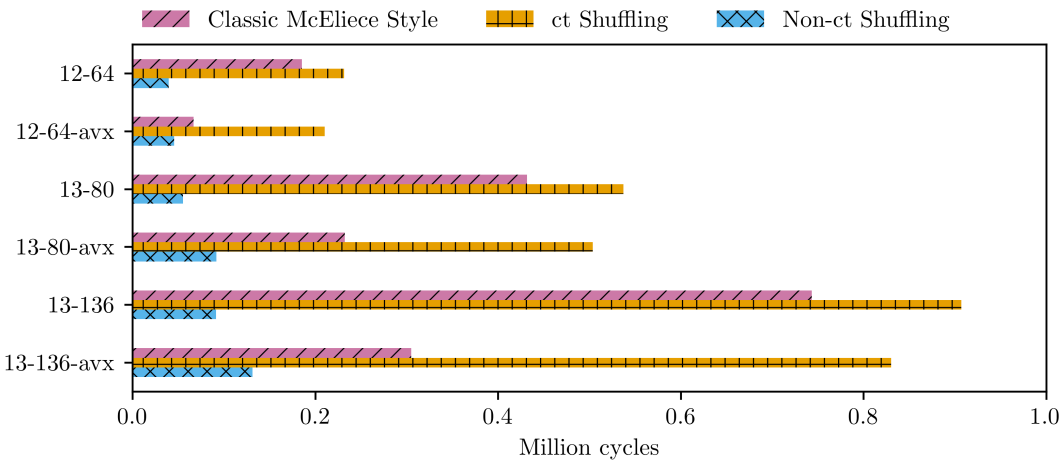


Figure 5.14. – Mean time spent creating the random error vector.

5.3.2. Encapsulation

As can be seen from Fig. 5.12, our new constant-time random error vector sampling makes up a significant amount of the total time, about 25-50 % depending on the parameters and AVX support. Such an increase is to be expected since our approach needs $\mathcal{O}(n \times \tau)$ time to avoid a cache-timing side-channel when setting the bits at the sampled indices. Additionally we can see that, as expected, the time needed for the matrix multiplication in the core encapsulation function increases significantly (see Fig. 5.13), since we don't skip operations when elements of the multiplied vector are zero. Thus our multiplication runs in time $\mathcal{O}(n)$ instead of $\mathcal{O}(\tau + \ell)$. From those runtime estimates we would predict an increase by a factor of somewhere around 12.8, 24.4 and 20.9 for the three different parameter-sets 12-64, 13-80, and 13-136 respectively. This roughly fits with our measurements, which give us factors of 9.1, 18.2 and 16.1 in the non-AVX2 case, and 16.2, 31.6 and 24.3 in the AVX2 case.

Creating the Random Error Vector

Figure 5.14 shows the time needed to create the random error vector using different approaches. While both constant-time versions are significantly slower than the non-ct version, they are relatively close to each other. Note, that for bit shuffling the Fisher-Yates shuffle performs considerably better than during key-generation above. This is due to the fact that we are only doing τ shuffles and accessing $\frac{n}{8}$ memory locations, leading to $\frac{n\tau}{8}$ memory accesses instead of n^2.

In the AVX2 version the Classic McEliece approach is considerably faster. This is due to less overhead when sampling the random numbers and vectorizeability of the loop that sets the bits. In contrast the Fisher-Yates shuffle is inherently sequential and can thus not be vectorized easily. This ability of the compiler to

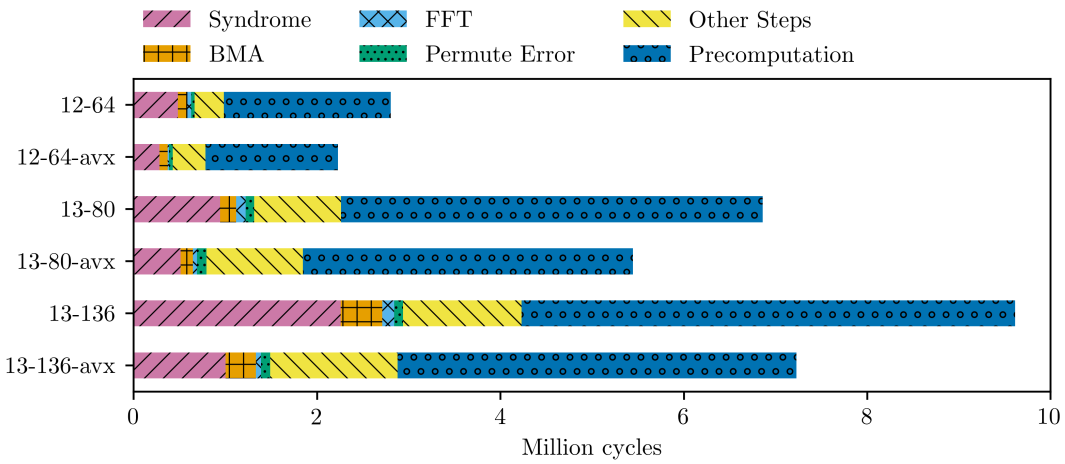


Figure 5.15. – Mean time spent in decapsulation for ct NTS-KEM.

optimize makes the Classic McEliece style approach clearly superior.

5.3.3. Decapsulation

Figure 5.15 gives an overview over the timings of the decapsulation steps. Since the only major change is the permutation of the error vector (and the precomputation of the controlbits), the remaining steps require the same time as in the non-constant-time version. As expected timings for the precomputation and permutation across the versions with $m = 13$ are roughly the same since the algorithms only depend on m and the timings roughly double compared to the 12-64-version, as expected from its $\mathcal{O}(n \log(n)^2)$ complexity. We can also see that the actual decapsulation given the controlbits is even slightly faster in the constant-time case than in the non-constant-time case. This can be explained by the fact that the original permutation is fully sequential, while the sorting-network-based approach is parallelized.

The time needed for the permutation in the AVX2 versions is the same as in the non-AVX2 version, since they use the same implementation. However we also see that this is justified, since the bottleneck clearly is syndrome computation followed by the reencapsulation step and the initial loading of the secret key data (both contained in the 'other steps').

Permuting Bitvectors

Here we have a look at the difference between our sorting network based bitvector permutation and the naive constant-time bitvector permutation. Figure 5.16 shows the time needed to permute the bitvector during decapsulation. We can immediately see two things.

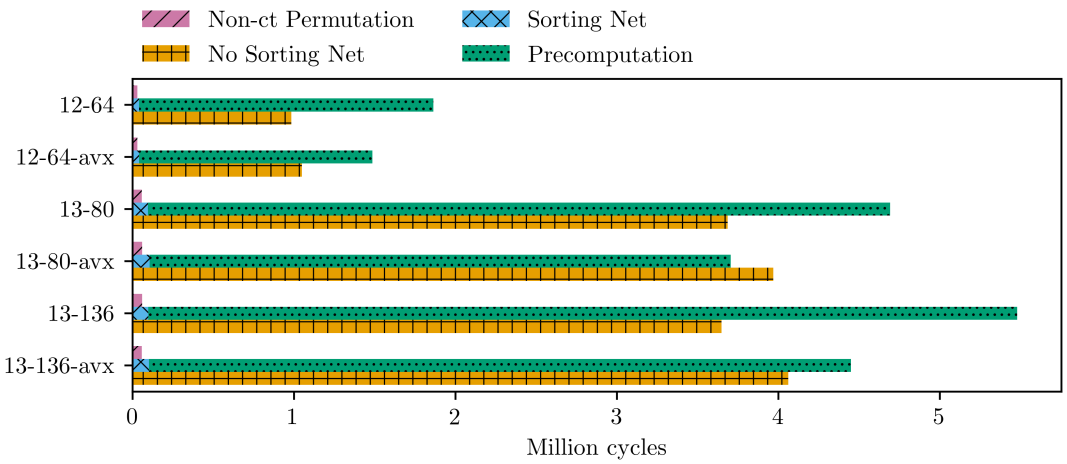


Figure 5.16. – Mean time spent permuting the bitvector during decapsulation.

Firstly, permuting using our sorting network is comparable in speed with the non-constant time permutation, when we ignore the cost of precomputation.

Secondly the cost of precomputation is immense and trumps even the cost of the naive ct permutation. Note however that in the avx2 versions there is still some space for optimization left, as the precomputation contains sequential parts that might be further parallelized, and the actual permutation is not using the AVX2 instructions which would allow 256 operations in parallel instead of 64. Especially the latter should be a comparatively easy change, though the performance gain is not required since, as noted above, the bottleneck is syndrome computation.

Looking at Fig. 5.17 we make a somewhat contradictory observation. Comparing with Fig. 5.15 we see that for the AVX2 versions the total time spent in decapsulation is lower in Fig. 5.15 even if including the precomputation, and that despite that permuting + precomputing is slower than the naive permutation. The cause seems to be the setup phase of the function, where the private key data is loaded into memory, which is significiantly faster in the version utilizing the sorting network. For example in the 13-136 AVX2 case we measured about 420 thousand cycles as mean setup time for the sorting network version and about 2 million cycles for the naive ct permutation. Note that this difference in setup time is also present when comparing the sorting network based version with the non-ct version and is independent of whether we use the AVX2 version or not, with the timings of the non-ct version being the same as the timings of the naive ct permutation. The direct cause of this is unknown. The only difference during setup is an additional null-check in the sorting network based version. Looking at the actual instructions in the assembled files, the null-check difference is also the only difference. It seems unlikely that this is the root cause however. More likely it might be due to differences in the instruction alignment in the assembly file or different cache access patterns.

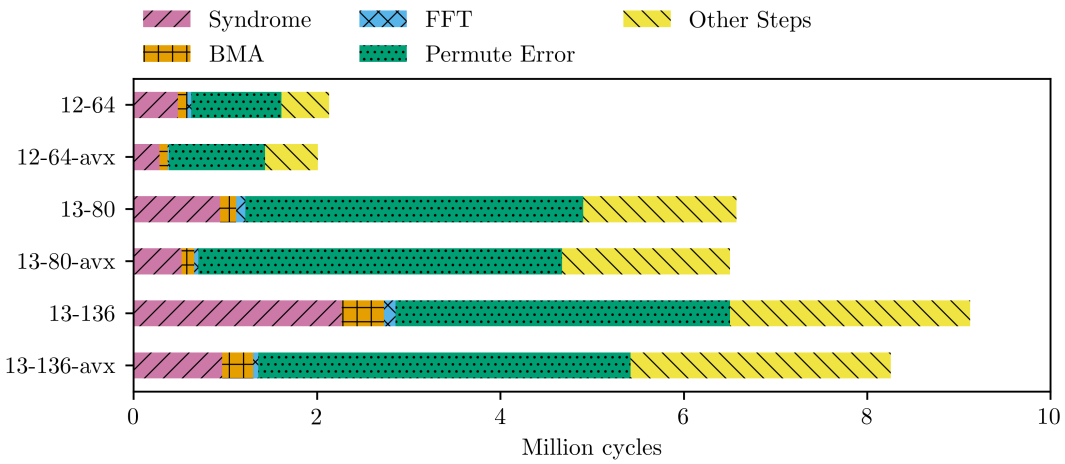


Figure 5.17. – Mean time in decapsulation using naive ct bitvector permutation.

Table 5.4. – Memory requirements in Bytes for sorting network based bitvector permutation.

	12-64	13-80	13-136
controlbits	73 728	173 056	173 056
masks	800	800	800
private-key size	328 736	947 316	1 439 626

Another aspect to consider are the memory requirements of this approach. For the most part our version should not consume significantly more memory than the non-constant-time version. However when using the sorting net with precomputed bits this is not the case, since we need to store the controlbits, as well as store some masks to select the bits. The memory requirements are presented in Table 5.4 together with the private key size, which is unchanged. On their own these memory requirements are rather high. However in the context of the private key size they seem more reasonable. Still, there is room for optimization. The space required by the masks could be halved, since we are essentially storing redundant masks. Moreover, based on Theorem A.2 we can calculate that the minimal amount of memory needed is 17 408 bytes for $m = 12$ and 40 960 bytes for $m = 13$. We are obviously far above that and optimization might be worthwhile. Note however that our current memory usage simplifies accessing the individual bits, and optimizations packing the data more tightly together most likely come with an increased runtime. Another alternative is the usage of a different permutation network. For example, the Beneš network needs about $n \log(n)$ comparisons, and thus would only require 6144 and 13 312 bytes to store the controlbits for the two different values of m.

Overall our sorting network based approach works very well. Especially when

we precompute and cache the controlbits, it performs even better than the original non-ct version, due to the above setup timings. There is room for memory usage optimization, however.

5.3.4. Sampling Bounded Integers

We now take a closer look at the three versions of sampling bounded integers we tried, namely Knuth-Yao, rejection sampling, and throwing away unnecessary bits when the bound is a power of two, which is the approach used in our ct version. For this we used these three different methods when sampling bounded integers during creation of the random error vector and look at the total time needed to create the random vector. The results are shown in Fig. 5.18 and are as expected. Throwing away the unused bits is fastest and Knuth-Yao is faster than rejection sampling, since it is an optimal way to sample arbitrarily bounded integers in contrast to rejection sampling. Rejection sampling is at least two times slower than throwing away unused bits in the AVX2 versions. This might be caused by loop vectorization, since we first do all the sampling and then throw away the unnecessary bits in one go, which can in theory be done in parallel. Interestingly our rejection sampling implementation is even 2 times slower than Knuth-Yao in the case of the 12-64 AVX2 version. Overall we can see that the choice of sampling method can have a very big impact on performance and time spent optimizing this can be worth the gains.

5.3.5. The Impact of Loop Vectorization

Lastly we will have a look at the impact of loop vectorization. As discussed in Section 4.6 we provide make targets to deactivate the compiler's loop vectorization. However deactivating it might have a potentially big impact on the performance since that decreases parallelism.

Figures 5.19 and 5.20 show our measurements. We omit the graph for key-generation, as there was no significant difference. In the non-AVX2 version everything behaves as we expect: the more we deactivate loop vectorization, the slower the execution gets. The effect is primarily visible for encapsulation, which stems from the fact that the random vector creation, which is vectorizeable, makes up for a significant part of the runtime.

Surprisingly disabling loop-vectorization globally in the AVX2 version performs better than local deactivation for encapsulation and actually improves performance in decapsulation. We can only speculate why this is the case and it is probably highly dependent on the compiler (version). It might be due to instruction alignment or force the compiler to use different optimizations that work better in our cases. However the differences are rather small. As such we don't think that disabling loop-vectorization globally should be done in general. However, it seems to be a valid option in case one cannot verify whether issues reported by ctgrind are actually false positives.

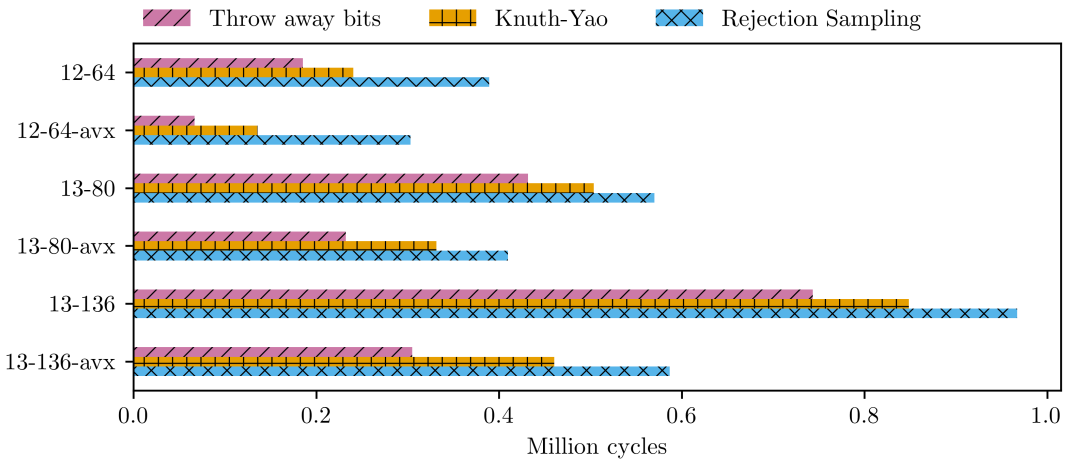


Figure 5.18. – Mean time in creation of random error **e** using different sampling methods.

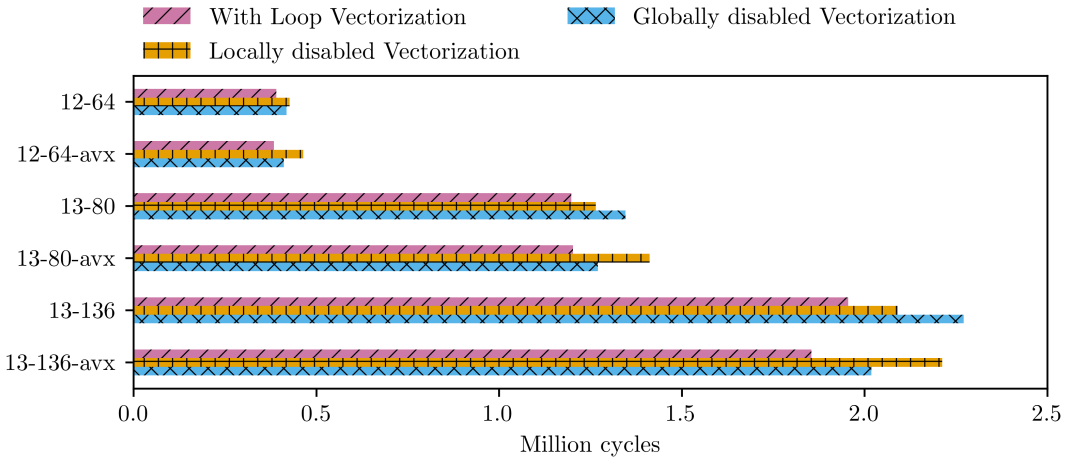


Figure 5.19. – Mean time for encapsulation for different loop vectorization settings.

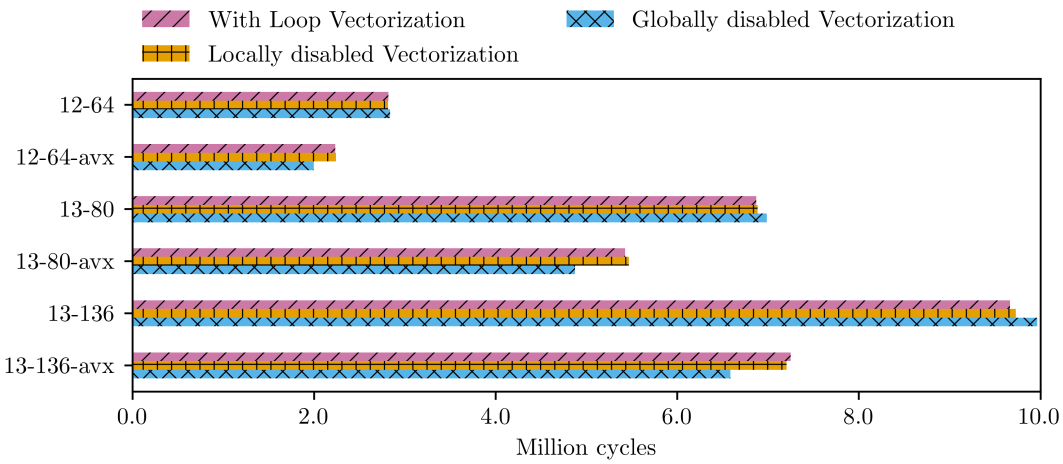


Figure 5.20. – Mean time for decapsulation for different loop vectorization settings.

5.3.6. Summary

The final version we picked is the ct in Table 5.1. This version uses the rejection sampling approach described in Section 4.2.3 to construct a systematic matrix. The random permutation is created using the sorting network approach from Section 4.5.1 using 64-bit integers and we also use sorting networks to apply the permutation during key-generation. During encapsulation we use the Classic McEliece approach to create the random error vector. For decapsulation we also use the sorting network approach with precomputed controlbits to permute the bitvectors. To uphold the API given by NIST we technically do not precompute the controlbits, instead our decapsulation function computes the controlbits and then calls a core decapsulation function which uses the controlbits. We do however extend the API to provide access to both the controlbit computing function as well as the core decapsulation function to allow users to cache the controlbits in their application. Based on our analysis this should provide the best performance, especially for decapsulation.

Lastly we want to remark on serialization. During (de-)serialization key data is transformed between a platform independent format used for key-distribution and a platform and implementation dependent format used to speedup computations. As we have seen through this evaluation, in all three functions (key-generation, encapsulation, decapsulation) (de-)serialization of key data has a significant impact on the total runtime. For key-generation there is not really much way to get around this, however for encapsulation and decapsulation this might be possible.

Under similar assumptions to the ones used when arguing for precomputation of the controlbits, namely that the same key is repeatedly used and thus only needs to be loaded once, it also seems plausible that this could be applied to serialization. This should be achievable by providing API access to deserialization functions and keeping the deserialized keys in memory for the lifetime of the pro-

gram. The encapsulation and decapsulation functions can then directly take the deserialized keys as inputs, and thus deserialization costs can be amortized across several invocations of encapsulation and decapsulation. We did not however implement this, and can therefore not provide concrete numbers on the performance gains. Sections 5.3.2 and 5.3.3 suggest significant improvements especially for encapsulation, though.

6. Conclusion

We have developed a constant-time implementation of NTS-KEM. We have seen that directly plugging-in constant-time code snippets can often lead to significant performance losses, especially when we are trying to prevent cache-timing attacks. In these cases we had to choose completely different approaches, often ones that are asymptotically not optimal when compared with the non-constant-time approaches. Moreover our performance analysis suggest that it can be highly beneficial to treat constant-timeness as an important requirement when designing a cryptographic scheme from the start, for example regarding the question of how to handle the controlbits.

We also saw that strict constant-timeness is sometimes not achievable. This is especially the case when dealing with randomized algorithms. We propose a relaxed security definition, which can be used for the analysis of algorithms and which we used several times.

As we have seen our implementation performs well, incurring a reasonable increase in encapsulation time, and, assuming per-key caching of the controlbits, a performance increase for decapsulation. Constant-time key-generation is slow, however. In practice this will not be an issue, though, since we do not expect NTS-KEM to be used in scenarios where lots of fresh keys have to be generated frequently. For the same reason we can also cache the controlbits to enable fast decapsulation.

For memory constrained systems, future implementations might want to look into optimizations of memory usage. However the large key-sizes already require a significant amount of memory. Thus there already is a substantial memory requirement given by the NTS-KEM specification, which we increase slightly.

We verified our implementation using ctgrind, a tool for checking constant-timeness based on valgrind's memcheck. Thus we can give users strong guarantees that our implementation does not contain timing-based side-channels.

A. Analysis of Odd-Even Mergesort

Theorem A.1. The number of calls to COMP by procedure Odd-Even-Merge in Algorithm 4.7 is

$$T_M(n) = \frac{n}{2}\left(\log(n) - 1\right) + 1 \tag{A.1}$$

Proof. The number of calls to COMP is given by the recursion:

$$T_M^{rec}(n) = 2 \cdot T_M^{rec}\left(\frac{n}{2}\right) + \frac{n}{2} - 1 \tag{A.2}$$

$$T_M^{rec}(2) = 1 \tag{A.3}$$

Now given $n = 2^k$, Eq. (A.1) is equivalent to:

$$T_M\left(2^k\right) = (k-1) \cdot 2^{k-1} + 1 \tag{A.4}$$

and Eq. (A.2) to:

$$T_M^{rec}\left(2^k\right) = 2 \cdot T_M^{rec}\left(2^{k-1}\right) + 2^{k-1} - 1 \tag{A.5}$$

We now prove equality of T_M^{rec} and T_M by induction over k, which concludes the proof:

Induction Hypothesis:

$$T_M^{rec}\left(2^{k-1}\right) = T_M\left(2^{k-1}\right) \tag{I.H.}$$

Base Case: $k = 1$

$$T_M\left(2^1\right) = 0 \cdot 2^0 + 1 = 1 = T_M^{rec}\left(2^1\right)$$

Step Case: $k - 1 \to k$

$$\begin{aligned}
T_M^{rec}\left(2^k\right) &= 2 \cdot T_M^{rec}\left(2^{k-1}\right) + 2^{k-1} - 1 \\
&\overset{\text{I.H.}}{=} 2 \cdot T_M\left(2^{k-1}\right) + 2^{k-1} - 1 \\
&= 2 \cdot \left((k-2) \cdot 2^{k-2} + 1\right) + 2^{k-1} - 1 \\
&= (k-2) \cdot 2^{k-1} + 2 + 2^{k-1} - 1 \\
&= (k-1) \cdot 2^{k-1} + 1 \\
&= T_M\left(2^k\right)
\end{aligned}$$

□

Theorem A.2. The number of calls to COMP by procedure ODD-EVEN-MERGESORT is

$$T_S(n) = \frac{n}{4} \cdot (\log(n) - 1) \cdot \log(n) + n - 1 \tag{A.6}$$

Proof. The number of calls to COMP is given by the recursion:

$$T_S^{rec}(n) = 2 \cdot T_S^{rec}\left(\frac{n}{2}\right) + T_M(n) = 2 \cdot T_S^{rec}\left(\frac{n}{2}\right) + \frac{n}{2}(\log(n) - 1) + 1 \tag{A.7}$$

$$T_S^{rec}(1) = 0 \tag{A.8}$$

By substituting $n = 2^k$ Eq. (A.6) is equivalent to:

$$T_S\left(2^k\right) = 2^{k-2} \cdot (k-1) \cdot k + 2^k - 1 \tag{A.9}$$

and Eq. (A.7) to:

$$T_S^{rec}\left(2^k\right) = 2 \cdot T_S^{rec}\left(2^{k-1}\right) + 2^{k-1}(k-1) + 1 \tag{A.10}$$

We prove equality of T_S^{rec} and T_S by induction over k.

Induction Hypothesis:

$$T_S^{rec}\left(2^{k-1}\right) = T_S\left(2^{k-1}\right) \tag{I.H.}$$

Base Case: $k = 0$

$$T_S\left(2^0\right) = 2^{-2} \cdot (-1) \cdot 0 + 2^1 - 1 = 1 - 1 = 0 = T_S^{rec}\left(2^0\right)$$

Step Case: $k - 1 \to k$

$$\begin{aligned}
T_S^{rec}\left(2^k\right) &= 2 \cdot T_S^{rec}\left(2^{k-1}\right) + 2^{k-1}(k-1) + 1 \\
&\overset{\text{I.H.}}{=} 2 \cdot T_S\left(2^{k-1}\right) + 2^{k-1}(k-1) + 1 \\
&= 2 \cdot \left(2^{k-3} \cdot (k-2) \cdot (k-1) + 2^{k-1} - 1\right) + 2^{k-1}(k-1) + 1 \\
&= 2^{k-2} \cdot (k-2) \cdot (k-1) + 2^k - 2 + 2^{k-1}(k-1) + 1 \\
&= 2^{k-2} \cdot (k-2) \cdot (k-1) + 2^{k-1}(k-1) + 2^k - 1 \\
&= (k-1) \cdot \left(2^{k-2} \cdot (k-2) + 2^{k-1}\right) + 2^k - 1 \\
&= 2^{k-2} \cdot (k-1) \cdot ((k-2) + 2) + 2^k - 1 \\
&= 2^{k-2} \cdot (k-1) \cdot k + 2^k - 1 \\
&= T_S\left(2^k\right)
\end{aligned}$$

□

B. The Security of the Knuth-Yao Algorithm

We claimed in Section 4.5 that the Knuth-Yao algorithm fulfills Definition 4.5.1 We will show this here.

Theorem B.1. The Knuth-Yao algorithm (Algorithm 3.3) fulfills Definition 4.5.1, where the secret $\mathcal{S}$ is the return value of the algorithm. That is, let X denote the random variable describing the return value of the Knuth-Yao algorithm, and T be the random variable describing its runtime. Then it holds that:

$$\Pr[X \mid T] = \Pr[X].$$

Proof. For this proof we use a Hoare-like notation to annotate the algorithm with logical formula F describing information about the execution state, here denoted as $\{|F|\}$. Each $\{|F|\}$ describes the state before (after) the execution of the next (previous) line of code. These are essentially invariants of the program, which hold at the given line. Logical entailment $\vdash$ is used to transform one formula into another. Note that when working within a probabilistic framework, a conditional branch involving a random variable corresponds to conditioning the distribution of the random variable on the truth value of the branch condition. This can be seen in Line 20 in Listing B.1, which shows the Knuth-Yao algorithm with our annotations.

Most of the steps can easily be read step by step. For the conditioning in Line 20, the following equations show the effect of conditioning on the distribution of x. Note that by Line 19 $X \sim \mathcal{U}(0, d+b-1)$.

$$\begin{aligned}
\Pr[x \mid x < d] &= \frac{\Pr[X = x \wedge X < d]}{\Pr[X < d]} = \frac{\Pr[\{x\} \cap \{0, 1, \ldots, d-1\}]}{\frac{d}{d+b}} \qquad (*) \\
&= \begin{cases} \frac{1}{d+b} \cdot \left(\frac{d}{d+b}\right)^{-1} = \frac{1}{d} & \text{if } 0 \leq x < d, \\ 0 & \text{otherwise.} \end{cases} \\
&\longrightarrow (x | x < d) \sim \mathcal{U}(0, d-1)
\end{aligned}$$

Listing B.1 – Knuth-Yao algorithm with Hoare annotations.

```
1   random_uint16_bounded(b){
2     d = 0; u = 1; x = 0;
3     {|x ∼ U(0, u − 1)|}
4     do{
5       {|(x ∼ U(0, u − 1)) ∨ (x ∼ U(0, d − 1) ∧ u = d)|}
6     ⊢ {|x ∼ U(0, u − 1)|}
7       while (u < b) {
8         {|x ∼ U(0, u − 1)|}
9         u = u << 1;
10        {|x ∼ U(0, u/2 − 1)|}
11        x = (x << 1) + randombit();
12        {|x ∼ U(0, u − 1)|}
13      }
14      {|x ∼ U(0, u − 1)|}
15      d = u - b;
16      {|x ∼ U(0, u − 1),  d = u − b|}
17    ⊢ {|x ∼ U(0, d + b − 1),  d = u − b|}
18      u = d;
19      {|x ∼ U(0, d + b − 1),  u = d|}
20    } while (x < d);
21    {|x ∼ U(d, d + b − 1)|}
22    x -= d;
23    {|x ∼ U(0, b − 1)|}
24    return x;
25  }
```

condition on $x < d^*$

condition on $x \geq d^{**}$

$$\Pr[x \mid x \geq d] = \frac{\Pr[X = x \wedge X \geq d]}{\Pr[X \geq d]} = \frac{\Pr[\{x\} \cap \{d, d+1, \ldots, d+b-1\}]}{\frac{b}{d+b}} \quad (**)$$

$$= \begin{cases} \frac{1}{d+b} \cdot \left(\frac{b}{d+b}\right)^{-1} = \frac{1}{b} & \text{if } d \leq x < d+b-1, \\ 0 & \text{otherwise.} \end{cases}$$

$$\longrightarrow (x|x \geq d) \sim \mathcal{U}(d, d+b-1)$$

Let X be the random variable denoting the return value, which is our secret, and T be the random variable denoting the runtime. Since all of the formulas in Listing B.1 are true, for every possible execution flow of the program and especially every number of possible loop iterations, we can conclude that for all possible execution times t the return value X is distributed uniformly over $\mathbb{N}_{b-1}$, i.e. $\forall t : \Pr[X \mid T = t] = \frac{1}{b}$. This implies $\Pr[X] = \frac{1}{b}$. □

C. Bugs found in NTS-KEM

During the work on this thesis, we found four bugs in the reference implementation of NTS-KEM. All of these bugs were reported to the NTS-KEM team and are fixed in the updated second round submission.

Wrong Hash Function

NTS-KEM uses a hash function to derive the encapsulated key. For this use of SHA3-256 is specified, the reference implementation used SHAKE-256 instead, which is a similar but different function of the Keccak family.

Use of Insecure memset

The reference implementation used memset to clean secrets from memory. However, use of memset in this scenario is not secure, since compilers are known to remove calls to memset as part of dead store elimination. We observed this happening with our compiler.

Malloc Failure not Handled

During checking the validity of the sampled Goppa polynomial the functions `formal_derivative_poly` and `gcd_poly` could return an error due to `malloc` failing. This failure was not handled by the caller which would have caused it to treat the polynomial as valid even though the check could not be performed.

Out of Bounds Array Access

The 13-80 and 13-136 AVX2 versions were violating array bounds when loading the ciphertext into vectorized AVX2 arrays.

Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. "An o(n Log n) Sorting Network." In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 1–9. ISBN: 0897910990. DOI: 10.1145/800061.808726.

[2] Martin R. Albrecht, Gregory V. Bard, and Clément Pernet. "Efficient Dense Gaussian Elimination over the Finite Field with Two Elements." In: *Computing Research Repository* abs/1111.6549 (2011). eprint: 1111.6549. URL: http://arxiv.org/abs/1111.6549.

[3] Martin Albrecht et al. *NTS-KEM*. URL: https://nts-kem.io/.

[4] Martin Albrecht et al. *NTS-KEM - Updated Second Round Submission*. available at https://nts-kem.io/. Nov. 2019.

[5] José Bacelar Almeida et al. "Jasmin: High-Assurance and High-Speed Cryptography." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078.

[6] José Bacelar Almeida et al. "Verifying Constant-Time Implementations." In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, 2016, pp. 53–70. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

[7] Jean-Philippe Aumasson. *Guidelines for low-level cryptography software*. URL: https://cryptocoding.net (visited on Jan. 29, 2020).

[8] Kenneth E. Batcher. "Sorting Networks and Their Applications." In: *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*. Vol. 32. AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., 1968, pp. 307–314. DOI: 10.1145/1468075.1468121. URL: https://doi.org/10.1145/1468075.1468121.

[9] Daniel J. Bernstein. *Cache-timing attacks on AES*. 2004. URL: http://cr.yp.to/papers.html#cachetiming.

[10] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. "McBits: fast constant-time code-based cryptography." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 610. URL: http://eprint.iacr.org/2015/610.

[11] Daniel J. Bernstein and Peter Schwabe. "New AES Software Speed Records." In: *Progress in Cryptology – INDOCRYPT 2008*. Vol. 5365. Lecture Notes in Computer Science. Springer, 2008, pp. 322–336. DOI: 10.1007/978-3-540-89754-5_25.

[12] Daniel J. Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion." In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.3 (2019), pp. 340–398. DOI: 10.13154/tches.v2019.i3.340-398.

[13] Daniel J. Bernstein et al. *Classic McEliece: conservative code-based cryptography - Second Round Submission*. available at https://classic.mceliece.org. Mar. 2019.

[14] Sunjay Cauligi et al. "FaCT: a DSL for timing-sensitive computation." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 174–189. DOI: 10.1145/3314221.3314605.

[15] Sunjay Cauligi et al. "FaCT: A Flexible, Constant-Time Programming Language." In: *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*. IEEE Computer Society, 2017, pp. 69–76. DOI: 10.1109/SecDev.2017.24.

[16] Jean-François Dhem et al. "A Practical Implementation of the Timing Attack." In: *Smart Card Research and Applications – CARDIS 1998*. Vol. 1820. Lecture Notes in Computer Science. Springer, 1998, pp. 167–182. DOI: 10.1007/10721064_15.

[17] Morris J. Dworkin et al. *Federal Inf. Process. Stds. (NIST FIPS) - 197: Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, Nov. 2001. DOI: 10.6028/NIST.FIPS.197.

[18] Shuhong Gao and Todd Mateer. "Additive Fast Fourier Transforms Over Finite Fields." In: *IEEE Transactions on Information Theory* 56.12 (Dec. 2010), pp. 6265–6272. ISSN: 0018-9448. DOI: 10.1109/TIT.2010.2079016.

[19] Daniel Genkin, Adi Shamir, and Eran Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Cryptology ePrint Archive, Report 2013/857. 2013. URL: https://eprint.iacr.org/2013/857.

[20] Benedikt Gierlichs et al. "Mutual Information Analysis." In: *Cryptographic Hardware and Embedded Systems – CHES 2008*. Ed. by Elisabeth Oswald and Pankaj Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Washington, DC, USA: Springer, Heidelberg, Germany, Aug. 2008, pp. 426–442. DOI: 10.1007/978-3-540-85053-3_27.

[21] David Leon Gil. *Keccak-Tiny*. URL: https://github.com/coruus/keccak-tiny (visited on Feb. 14, 2020).

[22] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. "ct-fuzz: Fuzzing for Timing Leaks." In: *CoRR* abs/1904.07280 (2019). arXiv: 1904.07280. URL: http://arxiv.org/abs/1904.07280.

[23] Michael Hutter and Jörn-Marc Schmidt. "The Temperature Side Channel and Heating Fault Attacks." In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 190. URL: http://eprint.iacr.org/2014/190.

[24] Raj Jain and Imrich Chlamtac. "The P^2 Algorithm for Dynamic Calculation of Quantiiles and Histograms Without Storing Observations." In: *Commun. ACM* 28.10 (1985), pp. 1076–1085. DOI: 10.1145/4372.4378.

[25] *Java SE 6 Update 17 Release Notes*. BugId 6863503. Oracle. URL: https://www.oracle.com/technetwork/java/javase/6u17-141447.html (visited on Jan. 29, 2020).

[26] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2007. ISBN: 978-1-58488-551-1.

[27] Donald E. Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN: 978-0-201-89685-5.

[28] Donald E. Knuth and Andrew C. Yao. "The Complexity of Nonuniform Random Number Generation." In: Donald E. Knuth. *Selected Papers on Analysis of Algorithms*. Vol. 102. CSLI lecture notes series. CSLI Publications, 2000, pp. 545–603. ISBN: 978-1-57586-212-5.

[29] Paul Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology – CRYPTO'96*. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[30] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Advances in Cryptology – CRYPTO'99*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[31] Hans Werner Lang. *Odd-even mergesort*. URL: https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm (visited on Feb. 23, 2020).

[32] Adam Langley. *ctgrind*. URL: https://github.com/agl/ctgrind.

[33] Varun Maram. *On the Security of NTS-KEM in the Quantum Random Oracle Model*. Cryptology ePrint Archive, Report 2020/150. 2020. URL: https://eprint.iacr.org/2020/150.

[34] Robert J. McEliece. *A Public-Key Cryptosystem Based On Algebraic Coding Theory*. Deep Space Network Progress Report. Jet Propulsion Laboratory, NASA, 1978. URL: https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF.

[35] Peter L. Montgomery. "Modular multiplication without trial division." In: *Mathematics of Computation* 44.170 (Apr. 1985), pp. 519–521. DOI: 10.1090/S0025-5718-1985-0777282-X.

[36] Elke De Mulder et al. "Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems." In: *Comput. Electr. Eng.* 33.5-6 (2007), pp. 367–382. DOI: 10.1016/j.compeleceng.2007.05.009.

[37] Harald Niederreiter. "Knapsack-type cryptosystems and algebraic coding theory." In: *Problems of Control and Information Theory 15*. 1986, pp. 159–166.

[38] NIST. *Post-Quantum Cryptography Standardization*. 2017. URL: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[39] D. Page. *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel*. Cryptology ePrint Archive, Report 2002/169. 2002. URL: https://eprint.iacr.org/2002/169.

[40] Thomas Pornin. *BearSSL – a smaller SSL/TLS library*. URL: https://bearssl.org.

[41] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "Dude, is my code constant time?" In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1697–1702. DOI: 10.23919/DATE.2017.7927267.

[42] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342.

[43] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. "Sparse representation of implicit flows with applications to side-channel detection." In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by Ayal Zaks and Manuel V. Hermenegildo. ACM, 2016, pp. 110–120. DOI: 10.1145/2892208.2892230.

[44] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 1095-7111. DOI: 10.1137/s0097539795293172. URL: https://arxiv.org/abs/quant-ph/9508027v2.

[45] *Valgrind*. URL: https://valgrind.org/.

[46] *Valgrind Documentation – Release 3.15.0*. Apr. 2019. URL: https://valgrind.org/docs/manual/valgrind_manual.pdf.

[47] Abraham Waksman. “A Permutation Network.” In: *J. ACM* 15.1 (1968), pp. 159–163. DOI: 10.1145/321439.321449.