DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking Supersingular Isogeny Diffie-Hellman Implementations

Jonas Hagg

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking Supersingular Isogeny Diffie-Hellman Implementations

Benchmarking Supersingular Isogeny Diffie-Hellman Implementierungen

Author: Jonas Hagg
Supervisor: Prof. Dr. Claudia Eckert
Advisor: Prof. Dr. Daniel Loebenberger
Submission Date: Submission date

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Submission date Jonas Hagg

Acknowledgments

Abstract

Kurzfassung

Contents

1. Background

This opening chapter covers the technical backgrounds needed to read and understand this thesis. Beside a short introduction into modern cryptography schemes, the emergence and consequences of quantum computers are explained. Finally, this chapter gives details about isogeny-based cryptography.

In modern cryptography one can differ between *symmetric* and *asymmetric* encryption schemes. While in a *symmetric* scheme the decryption and encryption of data is processed with the same key, *asymmetric* protocols introduce a key pair for every participant: A public key for encryption and a private key for decryption. The public key of *asymmetric* protocols is, as the name suggests, public to everyone. However, the private key needs to be secret and nobody but the producer may have knowledge about the private key.

Symmetric Cryptography

In Figure 1.1 a classical symmetric encryption scheme is shown. First, a plaintext is encrypted using a symmetric encryption algorithm and a secret key. The resulting chiffre text is transported to the receiver, where it is decrypted using the appropriate decryption algorithm and the same secret key. The red section in the middle represents an insecure channel (e.g. the internet), where attackers may read or modify data. Since for encryption and decryption the same secret key is used, the exchange of the key through the insecure channel is critical: Somehow the symmetric key needs to be transported securely to the receiver of the ciphertext.

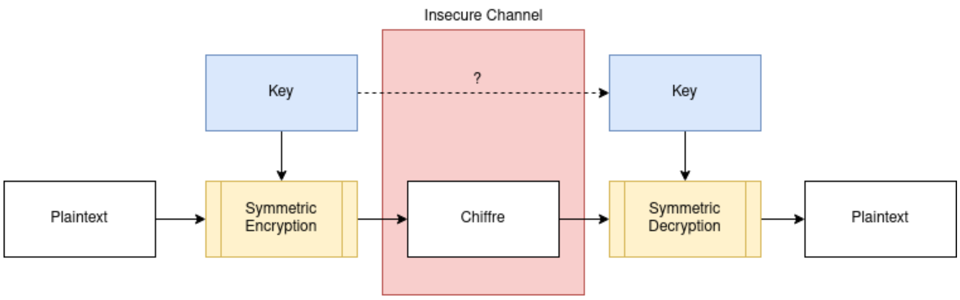

Figure 1.1.: Simple symmetric encryption scheme: Encryption and decryption algorithm use the same key.

In the following, the symmetric decryption and encryption is expressed in a more formal way. The common key k is used for encryption (Enc) and decryption (Dec), p is the plaintext and c is the ciphertext:

$$Enc(p,k) = c$$
$$Dec(c,k) = p$$

Asymmetric Cryptography

In asymmetric cryptography each participating subject needs to generate a key pair which consist of a private key and a public key. As mentioned above, the public key needs to be public (e.g. stored in a public database or a public key server). The private key, however, is only known to the subject and is kept secret. Figure 1.2 shows an example for asymmetric encryption. Assume Alice wants to send encrypted data to Bob. Therefore, Bob created a key pair and published his public key. Alice requests Bobs public key (e.g. from a public database) and uses it to encrypt the data. Once Bob received the ciphertext, he uses his secret private key for decryption in order to retrieve the original plaintext. In this work, the term *public-key encryption* or *public-key algorithm* is used as a synonym for asymmetric cryptography.

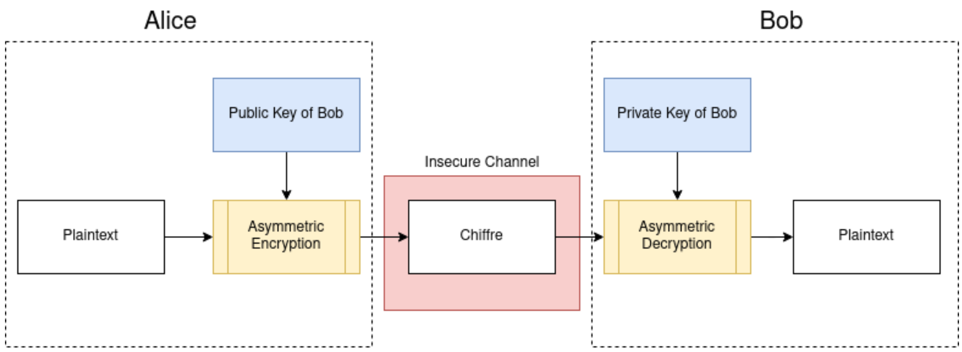

Figure 1.2.: Asymmetric encryption scheme: Encryption and decryption algorithm use different keys.

To formalize this procedure, again assume p as plaintext and c as ciphertext. The generated key pair of Bob consists of a private key for decryption (d_{Bob}) and a public key for encryption (e_{Bob}).

$$Enc(p, e_{Bob}) = c$$
$$Dec(c, d_{Bob}) = p$$

In contrast to the *symmetric* encryption, no secret key needs to be exchanged. However, the encryption and decryption of data using *asymmetric* encryption require intensive mathematical computations. Hence, the encryption of big sets of data using asymmetric encryption is not efficient.
One the other hand, *symmetric* encryption algorithms are usually based on simple operations,

such as bit shifting or XOR. This can be implemented efficiently in software and hardware. Thus, the practical relevance of *symmetric* encryption is enormous [1].
As stated above, securely exchanged keys are a precondition for the use of efficient *symmetric* encryption schemes. In order to exchange arbitrary keys securely, different key exchange protocols are available. The next section describes two basic protocols that establish a shared secret: The *Diffie-Hellman Key-Exchange* and a *Key Encapsulation Mechanism*.

1.1. Key Exchange

1.1.1. Diffie-Hellman Key-Exchange

The Diffie-Hellman key exchange was introduced by Whitfield Diffie and Martin Hellman in 1976 [2]. The protocol creates a shared secret between two participating subjects. The resulting shared key of the protocol is calculated decentralized and is never transported over an insecure channel.

Protocol

The classical Diffie-Hellman key exchange assumes, that Alice and Bobs want to create a shared secret key. Therefore, they agree on a big prime p and g, which is a primitive root modulo p[1]. Both, p and g are not secret and may be known to the public [3].

1. Alice choses a random $a \in \{1, 2, ..., p-2\}$ as private key.
2. Alice calculates the public key $A = g^a mod p$.
3. Bob choses a random $b \in \{1, 2, ..., p-2\}$ as private key.
4. Bob calculates the public key $B = g^b mod p$.
5. Alice and Bob exchange their public keys A and B.
6. Alice calculates:

$$\begin{aligned} k_{AB} &= B^a mod p \\ &= (g^b mod p)^a mod p \\ &= g^{ab} mod p \end{aligned} \tag{1.1}$$

7. Bob calculates:

$$\begin{aligned} k_{AB} &= A^b mod p \\ &= (g^a mod p)^b mod p \\ &= g^{ba} mod p \\ &= g^{ab} mod p \end{aligned} \tag{1.2}$$

[1]The primitive root modulo p is a generator element for the set S = $\{1, 2, ..., p-1\}$ [1].

8. Alice and Bob created the shared secret k_{AB}. Note that only the public keys of Alice and Bob were send over an insecure channel. The generated secret was calculated decentralized by Alice and Bob.

This procedure also can be illustrated in the following diagram, which emphasizes the commutative properties of the protocol. It does not make a difference, which function is applied first the starting point g ($x \to x^a$ or $x \to x^b$). The result is the same, since $g^{ab} = g^{ba}$.

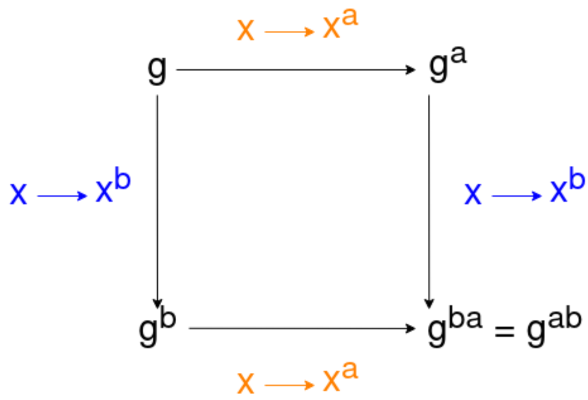

Figure 1.3.: Diffie Hellman diagram - both paths lead to the same result.

Security

If an attacker wants to compute k_{AB}, it needs to compute the private keys A and B of Alice and Bob. Since only the public keys are exchanged, the attacker needs to compute:

$$b = log_g\, B\, mod\ p$$
$$a = log_g\, A\, mod\ p$$

Hence, the security of the classical Diffie-Hellman key exchange is based on the discrete logarithm problem (see section 1.2). When using ephemeral key pairs, the Diffie-Hellman key exchange may be used as efficient perfect forward secrecy (PFS) protocol [1]. A protocol ensuring PFS does neither reveal any keys from the past nor future keys if an attacker compromises a arbitrary key of the protocol.

In modern cryptography elliptic curves are often used to increase the security of the Diffie-Hellman key exchange (ECDH). The participants have to agree on an elliptic curve and a point P on that curve. In order to generate a shared secret k_{AB} ECDH follows the same principles as described above. However, the protocol is adopted to work on elliptic curves. The advantage of ECDH is the increased security strength while using the same key size as classical Diffie-Hellman protocol [1].

Note that the introduced protocol does not authorize the participating subjects and does not guarantee integrity. Thus, this simple protocol may be exploited by a Man-In-The-Middle attack. A more advanced protocol using certificates and signed messages can be implemented, to guarantee authentication and integrity [1].

1.1.2. Key Encapsulation

A Key Encapsulation Mechanism (KEM) transmits a previously generated symmetric key to another subject. KEMs usually use asymmetric key pairs in order to encrypt the generated symmetric key. In the following the concept of KEMs is shown for the RSA based PKCS #1 v1.5 algorithm, which uses RSA key pairs to transmit a shared secret from Alice to Bob [4]:

1. Bob generates a RSA key pair (public key e_{Bob} and private key d_{Bob}) and transmits the public key to Alice.

2. Alice generates a random secret key k_{AB}:

$$k_{AB} = random()$$

3. Alice maps this secret to an integer m, using a well-defined mapping function h:

$$m = h(k_{AB})$$

4. Alice encrypts m with Bobs public key using the RSA encryption algorithm and transmits c to Bob.

$$c = RSA_{enc}(m, e_{Bob})$$

5. Bob decrypts the received ciphertext s to obtain the integer m:

$$m = RSA_{dec}(c, d_{Bob})$$

6. Finally bob uses the inverse mapping function h^{-1} to retrieve the shared secret:

$$k_{AB} = h^{-1}(m1)$$

1.1.3. Differences

Both presented key exchange primitives solve the challenge to securely share a symmetric encryption key between communication partners. However, there are a few differences between KEM and Diffie-Hellman. Firsty, while KEMs transmit a shared secret from one subject to another the calculation in the Diffie-Hellman protocol is done decentralized. Thus, the shared secret will never be send over an insecure channel.

Moreover, a KEM relies on a long term asymmetric key pair which is used to encapsulate and dencapsulate the randomly chosen shared secret. If the private key is compromised by an attacker all following symmetric encrypted communication could be revealed. On the contrary, a compromised Diffie-Hellman key exchange would only affect the messages which are encrypted using the secret resulting from that single Diffie-Hellman handshake. All following DH key exchanges are not compromised from the previously compromised exchange. In literature this is called *perfect forward secrecy (PFS)* [1].

1.2. Post-Quantum Cryptography

This section introduces the term *quantum computer* and describes its consequences on modern cryptography. In the following a *classical computer* refers to a non-quantum computer, which can be simulated by a deterministic Turing machine. In contrast to *classical computer* the term *quantum computer* describes a machine, which uses quantum mechanical phenomena to perform computations. It is important to note, that quantum computers can simulate classical computers [5]. In addition, classical computer are able to simulate quantum computers with exponential time overhead [5]. Thus, classical and quantum computations can calculate the same class of functions. However, quantum computers enable operations, that allow much faster computation [5].
In the past, scientists queried, if large-scale quantum computer are a physical possibility. It was stated, that the underlying quantum states are too fragile and hard to control. [6] Today, quantum error correction codes are known, which put a large-scale quantum computers within the realms of possibility [7]. However, it is still a big engineering challenge from a laboratory approach to a general-purpose quantum computer, which involves thousands or millions of physical qubits [6].

The security of modern asymmetric cryptographic primitives is usually based on difficult number theoretic problems, e.g. the discrete logarithm problem (DH, ECDH) or the factorization problem (RSA) [6]. These problems are theoretically solvable, but the computation of a result on classical computer claims an impractical amount of resources. In 2019, scientists solved the factorization problem for a 240 digit integer in about 900 core-years on a classical computer (one core year means running a CPU for a full year) [8]. In the following the discrete logarithm problem and the factorization problem are described.

Discrete Logarithm Problem

The discrete logarithm problem is the following challenge [9]: Given a prim p and two integers g and y. Find an integer x, such that

$$\begin{aligned} y &= g^x mod p \\ \iff x &= log_g\, y\, mod p \end{aligned}$$

Until today, it is not known if a classical computer is able to compute the general discrete logarithm problem in polynomial time. Thus, the discrete logarithm problem is considered to be difficult so solve for classical computers [9]. This assumption makes the discrete logarithm problem an attractive basis for various cryptographic primitives: DSA, ElGamal, classical Diffie-Hellman, and elliptic curve Diffie-Hellman (ECDH) exploit the hardness of the discrete logarithm problem in order to secure their algorithms.

Factorization Problem

Given two large primes p and q, it is easy to compute the the product of them:

$$n = p \star q$$

For a given n, however, it is difficult to find the prime factors p and q. The computation of the prime factorization for a given integer n is called the factorization problem [1]. For large numbers n no efficient algorithm for classical computers is known to solve this challenge [1]. The most famous cryptographic protocol, which builds upon the hardness of the factorization problem is RSA.

1.2.1. Impact of Quantum Computers on Cryptography

As stated above, quantum computers enable new operations which speed up certain algorithms. Two quantum algorithms which have enormous consequences on modern cryptography are *Shor's algorithm* and *Grover's algorithm* [5].

Shor's Algorithm

Peter Shor published *"Algorithms for quantum computation: discrete logarithms and factoring"* in 1994 [10], where he demonstrated that the factorization problem and the discrete logarithm problem can be solved in polynomial time on quantum computers. Both problems are the basis of many public-key systems (RSA, DH, ECDH, ...), which are used intensively in modern communication systems. Hence, a quantum computer running *Shor's algorithm* would qualify the assumption of most asymmetric encryption schemes and thus break their security.

Grover's Algorithm

The second algorithm having impact on computer security was published by Lov Grover in 1996 (*"A fast quantum mechanical algorithm for database search"*, [11]) - namely *Grover's algorithm*. The algorithms solves the problem of finding an element y in a set s (e.g. a database) where $|s| = N$. On a classical computer an algorithm solving this problem runs in $\mathcal{O}(N)$, however *Grover's algorithm* has complexity $\mathcal{O}(\sqrt{N})$ [5].
In contrast to public-key systems, which relay on hard mathematical problems, symmetric encryption schemes relay on the secrecy of a randomly generated key. Thus, to break symmetric encryption one need to perform a brute-force attack on the symmetric key. Using *Grover's algorithm* offers a square root speed up on classical brute force attacks [12]. Assume a randomly generated n-bit key. A classical brute force algorithm lies in $\mathcal{O}(2^n)$, which is considered to be safe for a big n (e.g. n=128). *Grover's algorithm* speeds up this attack to $\mathcal{O}(\sqrt{2^n}) = \mathcal{O}(2^{n/2})$ [12]. However, the complexity is still exponential and with a growing key size n the security can be further increased. Thus, *Grover's algorithm* forces symmetric encryption schemes to increase their key size in order to stay secure.

To sum up, quantum computers make use of quantum mechanical phenomena in order to solve mathematical problems, which are assumed to be difficult for modern computers. As a result large-scale quantum computers might break many algorithms of modern *asymmetric* cryptography and enforce increased key sizes for *symmetric* encryption schemes. The following table form the NIST *"Report on Post-Quantum Cryptography"* [6] shows the impact of quantum computers on modern encryption schemes:

Cryptographic algorithm	**Type**	**Purpose**	**Impact from quantum computer**
AES	Symmetric key	Encryption	Larger key sizes needed
SHA	—	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
DSA	Publiy key	Signatures, key exchange	No longer secure
ECDH, ECDSA	Public key	Signatures, key exchange	No longer secure

Table 1.1.: Impact of quantum computers on modern encryption schemes (adopted from [6]).

In contrast to this development in modern cryptography NIST states [6]:

> *"In the last three decades, public key cryptography has become an indispensable component of our global communication digital infrastructure. These networks support a plethora of applications that are important to our economy, our security, and our way of life, such as mobile phones, internet commerce, social networks, and cloud computing. In such a connected world, the ability of individuals, businesses and governments to communicate securely is of the utmost importance."*

This statement emphasizes the urgency and need of new asymmetric encryption schemes. As a consequence, NIST initiated a process to standardize quantum-secure public-key algorithms. In literature this is called *post-quantum cryptography* since the objectives of the submitted procedures is to stay secure against a large-scale quantum computer. In July 2020, Round 3 of the standardization process was announced. Different approaches for quantum-resistant algorithms have been proposed. In this work, the focus is on isogeny-based cryptography (section 1.3). Other classes of post-quantum cryptography are described for completeness in the following section 1.2.2.

1.2.2. Classes of Post-Quantum Cryptography

This section provides an overview about important post-quantum cryptography classes: Lattice-based, multivariate and code-based cryptography as well as hash-based signatures are shortly presented.

Lattice-based Cryptography

Lattice-based cryptography is - as the name suggests - based on the mathematical construct of lattices[2]. There are different computational optimization problems involving lattices, which are considered to be hard to solve even for quantum computers [13]. In 1998, NTRU was published as the first public-key system based on lattices [14]. Since then NTRU was continuously improved resulting in NTRUencrypt (public-key system) and NTRUsign (digital signing algorithm). Furthermore, a fully homomorphic encryption scheme based on lattices was published in 2009 [15].
Lattice-based cryptography is characterized by simplicity and efficiency [6]. The security of existing implementations (NTRU or Ring-LWE) can be reduced to NP-hard problems. However, lattices encryption schemes have problems to prove security against known cryptoanalysis [6].

Multivariate Cryptography

Multivariate cryptography are public key systems that are based on multivariate polynomials (e.g. $p(x, y) = x + 2y$) over a finite field $\mathbb{F}$. Their security is based on the prove, that solving systems of multivariate polynomials are NP-hard [16]. This makes multivariate public key systems attractive for post-quantum cryptography; especially their short signatures make them a candidate for quantum-secure digital signature algorithms [17], e.g. the Rainbow signature scheme [18].

Code-based Cryptography

Code-based cryptographic primitives are build upon error-correcting codes. A public key system using error-correcting codes uses a public key to add errors to a given plaintext resulting in a ciphertext. Only the owner of the private key is able to correct these errors and to reconstruct the plaintext [19]. McEliece, published in 1978, was the first of those systems and it has not been broken until today [20]. On the other hand, code-based cryptography requires large key sizes [19].
Beside asymmetric cryptography, code-based schemes have been proposed for digital signatures, random number generators and cryptographic hash functions [19].

Hash-based Signatures

Hash-based signatures describe the construction of digital signatures schemes based on hash functions. Thus, the security of theses primitives is based on the security of the underlying hash function and not on hard algorithmic problems [19]. Since hash functions are widely deployed in modern computer systems, the security of hash-based signatures is well understood [6].
The initially developed One-Time Signatures has the downside, that a new public key pair

[2] A lattice l is a subgroup of $\mathbb{R}^n$. In the context of cryptography usually integer lattices are considered: $l \subseteq \mathbb{Z}^n$ [13].

is needed for each signature [21]. In 1979, Merkle introduced the Merkle Signature Scheme (MSS), which uses one public key for multiple signatures [22]. Further improvements of MSS introduced public keys, which can be used for 2^{80} signatures. However, this also leads to longer signature sizes [21].

1.3. Isogeny-based Cryptography

Isogeny-based cryptography was proposed in 2011 as a new cryptographic system, that might resists quantum computing [23]. Beside the publication describing isogeny-based cryptgraphy, the authors also provided a reference implementaion of a public key system called *SIKE* [24]. Isogeny-based cryptography benfits from small key sizes compared to other post-quantum cryptography classes, however, their performance is comparatively slow [24]. The security of these primitives is based on finding isogenies between supersingular elliptic curves.
In the following, the problem is roughly illustrated. It is not indented to precise the mathematics behind isogeny-based cryptography, since this is not the scope of this work. However, the reader might become a little understanding of the magic behind supersingular isogenies. Then, the central component of isogeny-based cryptography - namely the Supersingular Isogeny Diffie Hellman (SIDH) - is described. Afterwards, details of the reference implementation SIKE are given and the security of SIDH is considered.

1.3.1. Illustration of the Problem

This section is adopted from [25] and [26].
Isogeny-based cryptography works on supersingular elliptic curves. To be more precise: It is based on isogenies between supersingular elliptic curves.
In the context of elliptic curves one can calculate a quotient of an elliptic curve E by a subgroup S. This essentially means to construct a new elliptic curve $E\backslash S$. Beside this new curve, the procedure also yields a function $\phi_S : E \rightarrow E\backslash S$ which is called *isogeny*. Carefully chosen elliptic curves E have a wide range of subgroups which can be used to construct many isogenies.
Supersingular elliptic curves are a special type of elliptic curves having properties that are useful for cryptography. Since supersingular elliptic curves can be seen as subset from ordinary elliptic curves, they can also be used to calculate isogenies between them, as described above.
The idea behind isogeny-based cryptography might be illustrated as followed:

1. Start with a known curve E and build a isogeny to a arbitrary reachable curve E_A.

2. This yields the isogeny $\phi_A : E \rightarrow E_A$, which is used as private key.

3. The curve E_A is used as part of the public key.

As usually in asymmetric cryptography the hard mathematical problem is the calculation of the private key while knowing the public key. To be more precise: Find the isogeny

$\phi_A : E \rightarrow E_A$ while knowing curves E and E_A is considered to be a quantum-resistant challenge. In literature, this is formally defined as *SIDH problem* [24].
This key pair, however, is not used to decrypt or encrypt data. In fact, the procedure is very similar to the previously introduced Diffie-Hellman key exchange where the key pair is used to establish a shared secret between two communication partners. In its core, isogney-based cryptography creates a shared secret via a Diffie-Hellman like procedure. This is called *Supersingular Isogeny Diffe Hellman (SIDH)*.

1.3.2. Supersingular Isogeny Diffie Hellman (SIDH)

Recall the Diffie Hellman diagram in Figure 1.3 where the commutative property of the protocol was shown. The same diagram can be drawn for the Diffie Hellman on supersingular isogenies (see Figure 1.4):

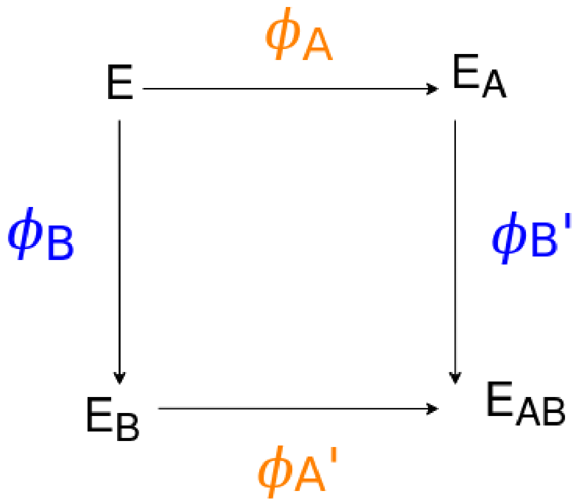

Figure 1.4.: Supersingular Isogeny Diffie Hellman procedure - both paths lead to the same result. Note the different isogenies ϕ'_A and ϕ'_B, that are applied in each second step of the diagram. The reason for this lies in the mathematics of supersingular isogenies: $\phi_A\phi_B \neq \phi_B\phi_A$. In order to construct ϕ'_X or ϕ'_B the public key of each communication partner provides additional information beside the curve E_A or E_B. [27]

In the previous section the underlying mathematical idea was illustrated. The described key generation can be extended to a key exchange primitive which has strong similarity to the classical Diffie-Hellman key exchange. Starting point of SIDH is a known supersingular elliptic curve E.

1. Alice creates an isogeny ϕ_A (private key) which leads to curve E_A (part of the public key).

2. Bob creates an isogeny ϕ_B (private key) which leads to curve E_B (part of the public key).
3. Alice and Bob exchange their public keys.
4. Bob computes ϕ'_B (using additional information from the public key of Alice) and applies ϕ'_B to the received E_A. This results in E_{AB}
5. Alice computes ϕ'_A (using additional information from the public key of Bob) and applies ϕ'_A to the received E_B. This results in E_{AB}
6. Alice and Bob now share the common secret E_{AB}.

1.3.3. Implemenation Details

The reference implementation of SIKE [24] provides two fundamental functions: *isogen* and *isoex*. Both are used, to implement the previously introduced Supersingular Isogeny Diffie-Hellman (SIDH) algorithm. Note that the secret key *sk* is represented as a random integer.

Function	Input	Output
isogen	secret key *sk*	public key *pk*
isoex	secret key *sk*, public key *pk*	shared secret *sec*

Table 1.2.: Core functions of the SIKE reference implementation.

The function *isogen* takes a secret key (random integer) as input an generates the public key. The shared secret is generated by *isoex* taking the own secret key and the foreign public key as input. The key exchange procedure with respect to *isogen* and *isoex* works as followed:

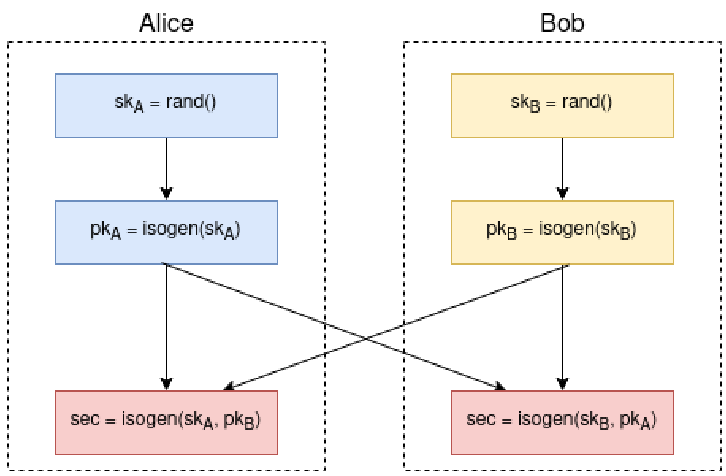

Figure 1.5.: SIDH based on *isogen* and *isoex*: After generating a random secret key *sk* each party computes its public key *pk*. After the exchange of public keys each party finally calculates the shared secret *sec*.

Beside this key exchange algorithm, SIKE provides a complete asymmetric encryption scheme and a key encpsulation mechansim [24]. Both of these schemes build upon the here described SIKE core functions *isogen* and *isoex*.

Isogeny-based PKE

The isogeny-based public key encryption system (PKE, Figure 1.6) consists of three algorithms:

1. *Gen* generates a key pair *(sk, pk)*.
2. *Enc* encrypts a given plaintext m using a foreign public key pk and the own secret key r.
3. *Dec* decodes a given cipthertext using the own secret key sk

Note, that the function F used in *Enc* and *Dec* is a key derivation function.

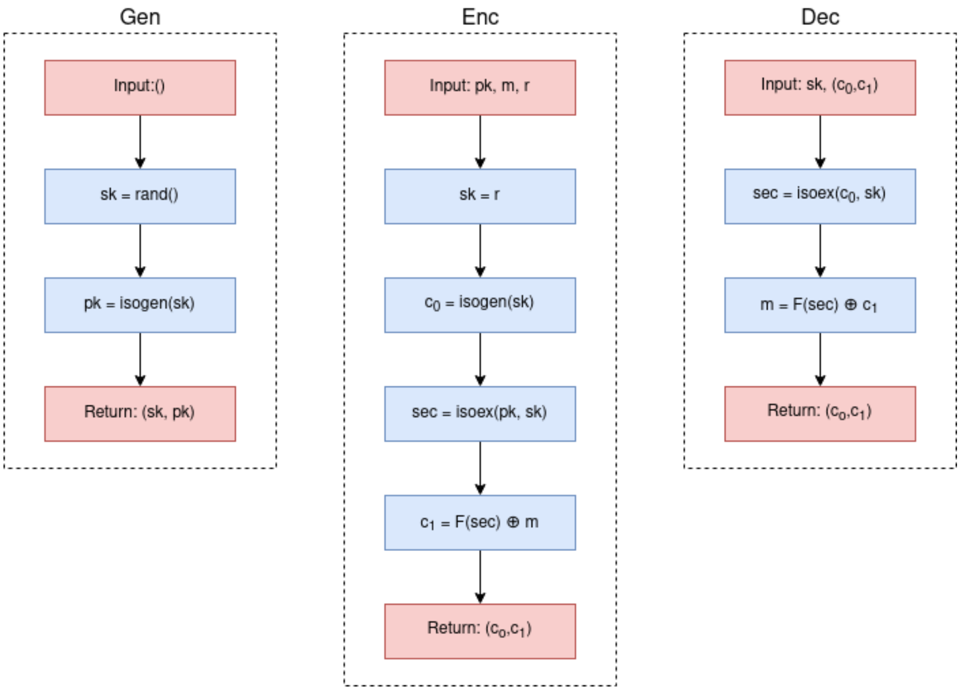

Figure 1.6.: Isogeny-based public key encryption (PKE) scheme.

Isogeny-based KEM

The isogeny-based key encapsulation mechanism (KEM, Figure 1.7) consists of three algorithms:

1. *Gen* generates a key pair *(sk, pk)* and a secret s.
2. *Ecnaps* takes a given public key as input and calculates a secret to share named K. Moreover the function returns a ciphertext c, that will be forwarded to the owner of the public key.

3. *Dec* takes a ciphertext c and the output of *Gen* as input in order to retrieve the shared secret K.

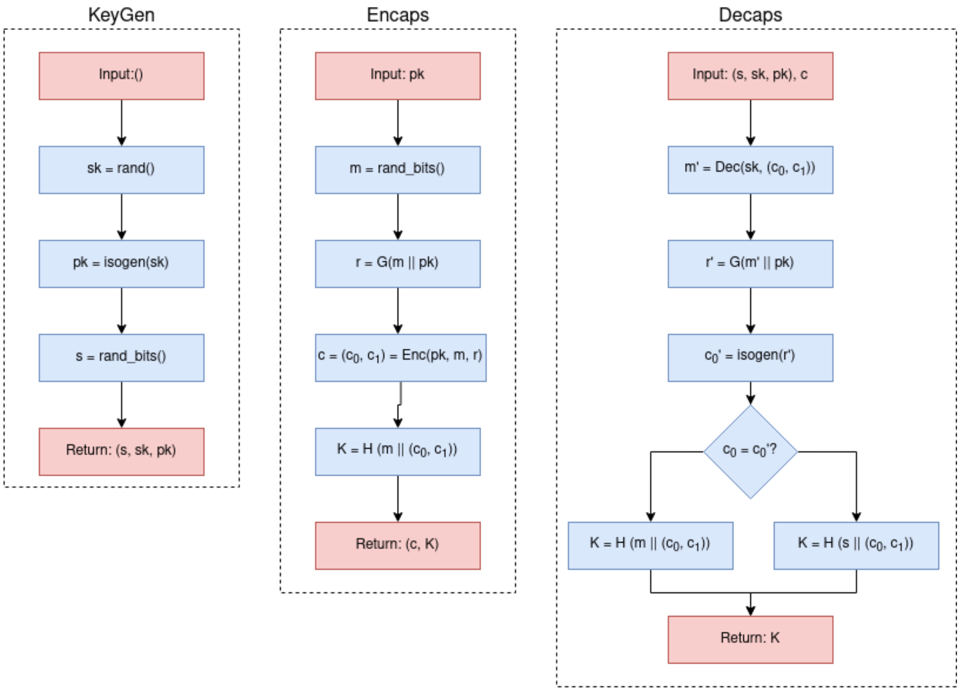

Figure 1.7.: Isogeny-based key encapsulation mechanism (KEM).

1.3.4. Security considerations

The security of isogeny-based cryptography is based on the hardness of the *SIDH problem*: Given two supersingular elliptic curves E and E', find an isogeny between them [24].
The SIKE reference implementation proposes different parameter sets, each supposing to ensure a NIST defined security level. These NIST defined security levels are:

1. Any attack breaking this security level must require resources comparable to perform a key search on a 128-bit key (e.g. AES128).

2. Any attack breaking this security level must require resources comparable to perform a collision search search on a 256-bit hash function (e.g. SHA256).

3. Any attack breaking this security level must require resources comparable to perform a key search on a 192-bit key (e.g. AES192).

4. Any attack breaking this security level must require resources comparable to perform a collision search search on a 384-bit hash function (e.g. SHA384).

5. Any attack breaking this security level must require resources comparable to perform a key search on a 256-bit key (e.g. AES256).

The proposed parameter sets of SIKE are named after the bit length of the underlying primes:

- `SIKEp434` supposed to satisfy NIST securiy level 1 (AES128)
- `SIKEp503` supposed to satisfy NIST securiy level 2 (SHA256)
- `SIKEp610` supposed to satisfy NIST securiy level 3 (AES192)
- `SIKEp751` supposed to satisfy NIST securiy level 5 (AES256)

Current research provides confidence that the SIKE parameter sets satisfy the defined security levels even under the assumption of currently known algorithms [28]. Therefore, the authors consider three algorithms to solve the *SIDH problem*: *Tani's quantum claw finding algorithm* [29], *Grover's algorithm* [11] and a *parallel collision-finding algorithm* [30].

Ephemeral SIDH keys are predestined to implement quantum-secure perfect forward secrecy protocols (PFS) [31]. PFS ensures that a compromised long-term key does not reveal past or future keys of the protocol.

Side-channel attacks against isogeny-basd cryptography might 1) reveal parts of the secret private key or 2) reveal parts of the public key computation. To protect against power-analysis side-channel attacks it is recommend to prefer constant-time implementations [24]. The authors state, however, that an attacker has *"access to a wide range of power, timing, fault and various other side-channels"*. Thus, preventing isogeny-based cryptography from all side-channel attacks seems to be nearly impossible.

2. Description of existing SIDH implementations

Currently, three implementations of Supersingular Isogeny Diffie-Hellman (SIDH) are available, namely *SIKE*, *CIRCL* and *PQCrypto-SIDH*. In this chapter each implementation is introduced in detail. At the end of the chapter, Table 2.1 shows similarities and differences between all approaches.
In the following, some algorithms are described as *compressed*. These compressed version exploit shorter public key sizes while increasing the computation time of the algorithms.

2.1. SIKE

SIKE stands for **S**upersingular **I**sogeny **K**ey **E**ncapsulation. It is the reference implementation of the first proposed isogeny-based cryptographic primitives [23]. Today, SIKE is a NIST candidate for quantum-resist *"Public-key Encryption and Key-establishment Algorithms"*. It is developed by a cooperation of researchers, lead by David Jao [24].
SIKE implements its key encapsulation mechanism (KEM) upon a public key encryption system (PKE), which is built upon SIDH (as highlighted in chapter 1). Beside a generic reference implementation, SIKE offers various optimized implementations of their cryptographic primitves:

- Generic optimized implementation, written in portable C
- x64 optimized implementation, partly written in x64 assembly
- x64 optimized compressed implementation, partly written in x64 assembly
- ARM64 optimized implementation, partly written in ARMv8 assembly
- ARM Cortex M4 optimized implementation, partly written in ARM thumb assembly
- VHDL implementation

All of these implementations can be run with the following parameter sets: `p434`, `p503`, `p610` and `p751`. SIKE states to countermeasure timing and cache attacks by implementing constant time cryptography [24].

SIKE API

The API of SIKE for a SIDH key exchange is the following:

```
// Generate random private key for Alice
random_mod_order_A(PrivateKey_A);

// Generate random private key for Bob
random_mod_order_B(PrivateKey_B);

// Generate ephemeral public key for Alice
EphemeralKeyGeneration_A(PrivateKey_A, PublicKey_A);

// Generate ephemeral public key for Bob
EphemeralKeyGeneration_B(PrivateKey_B, PublicKey_B);

// Computation of shared secret by Alice
EphemeralSecretAgreement_A(PrivateKey_A, PublicKey_B, SharedSecret_A)

// Computation of shared secret by Bob
EphemeralSecretAgreement_B(PrivateKey_B, PublicKey_A, SharedSecret_B)
```

All parameters used in this API are of type `unsigned char*`. Note that for all implementations and all parameter sets the API is the same. Therefore, during compilation one need to include the correct files to initialize SIKE with a specific parameter set.

2.2. PQCrypto-SIDH

PQCrypto-SIDH is a software library mainly written in C. It is developed by Microsoft for experimental purposes [32]. Note, that many developers of SIKE also work for Microsoft leading to great similarities between SIKE and PQCrypto-SIDH. However, in terms of compression, SIKE references the here described Microsoft library in its documentation.
The PQCrypto-SIDH library implements a isogney-based KEM and the underlying SIDH. Moreover, the library offers the following optimized versions:

- Generic optimized implementation, written in portable C
- Generic optimized compressed implementation, written in portable C
- x64 optimized implementation, partly written in assembly
- x64 optimized compressed implementation, partly written in assembly
- ARMv8 optimized implementation, partly written in assembly
- ARMv8 optimized compressed implementation, partly written in assembly

All of these implementations can be run with the following parameter sets: `p434`, `p503`, `p610` and `p751`. The developers argue to protect the algorithms against timing and cache attacks. Therefore, the library implements constant time operations on secret key material [32].

PQCrypto-SIDH API

The API of PQCrypto-SIDH for a SIDH key exchange is the following:

```
// Generate random private key for Alice
random_mod_order_A_SIDHpXXX(PrivateKey_A);

// Generate random private key for Bob
random_mod_order_B_SIDHpXXX(PrivateKey_B);

// Generate ephemeral public key for Alice
EphemeralKeyGeneration_A_SIDHpXXX(PrivateKey_A, PublicKey_A);

// Generate ephemeral public key for Bob
EphemeralKeyGeneration_B_SIDHpXXXPrivateKey_B, PublicKey_B);

// Computation of shared secret by Alice
EphemeralSecretAgreement_A_SIDHpXXX(PrivateKey_A, PublicKey_B, SharedSecret_A)

// Computation of shared secret by Bob
EphemeralSecretAgreement_B_SIDHpXXX(PrivateKey_B, PublicKey_A, SharedSecret_B)
```

For XXX $\in$ {`434, 503, 610, 751`}. All parameters used in this API are of type `unsigned char*`.

2.3. CIRCL

CIRCL (**C**loudflare **I**nteroperable, **R**eusable **C**ryptographic **L**ibrary) is a by Cloudflare developed collection of cryptographic primitives [33]. CIRCL is written in Go and implements some quantum-secure algorithms like SIDH and an isogeny-based KEM. Cloudflare does not guarantee for any security within their library. Furthermore, the isogeny-based cryptographic primitives are adopted from the official SIKE implementation. The following implementation optimizations are stated to be available:

- Generic optimized implementation, written in Go (unfortunately, this version could not be compiled)
- AMD64 optimized implementation, partly written in assembly
- ARM64 optimized implementation, partly written in assembly

Note, that there are no compressed versions available. The library supports the following parameter sets: p434, p503 and p751. To avoid side-channel attacks, their code is implemented in constant time [34].

CIRCL API

The API of CIRCL for a SIDH key exchange is the following:

```
// Generate random private key for Alice
PrivateKey_A = sidh.NewPrivateKey(sidh.FpXXX, sidh.KeyVariantSidhA)
PrivateKey_A.Generate(rand.Reader)

// Generate random private key for Bob
PrivateKey_B = sidh.NewPrivateKey(sidh.FpXXX, sidh.KeyVariantSidhB)
PrivateKey_B.Generate(rand.Reader)

// Generate public key for Alice
PublicKey_A = sidh.NewPublicKey(sidh.FpXXX, sidh.KeyVariantSidhA)
PrivateKey_A.GeneratePublicKey(PublicKey_A)

// Generate public key for Bob
PublicKey_B = sidh.NewPublicKey(sidh.FpXXX, sidh.KeyVariantSidhB)
PrivateKey_B.GeneratePublicKey(PublicKey_B)

// Computation of shared secret by Alice
SharedSecret_A := make([]byte, PrivateKey_A.SharedSecretSize())
PrivateKey_A.DeriveSecret(SharedSecret_A, PublicKey_B)

// Computation of shared secret by Bob
SharedSecret_B := make([]byte, PrivateKey_B.SharedSecretSize())
PrivateKey_B.DeriveSecret(SharedSecret_B, PublicKey_A)
```

For XXX $\in$ {434, 503, 751}.

2.4. Overview

	SIKE	PQCrypto-SIDH	CIRCL
Developer	Research cooperation	Microsoft	Cloudflare
Language	C Assembly	C Assembly	GO Assembly
Reference	C Assembly	C Assembly	GO Assembly
Implemented primitives	SIDH PKE KEM	SIDH KEM	SIDH KEM
Available parameters	p434 p503 p610 p751	p434 p503 p610 p751	p434 p503 p751
Optimized versions	Generic (portable c) x64 x64 compressed ARM64 ARM Cortex M4 VHDL	Generic (portable c) x64 x64 compressed ARMv8 ARMv8 compressed	Generic (GO) AMD64 ARM64
Security	Constant time	Constant time	Constant time

Table 2.1.: Overview and comparison of existing SIDH implementations.

3. Benchmarking Suite

In this chapter the SIDH implementations introduced in chapter 2 are compared with each other. For this purpose, a benchmarking suite was developed, which allows the generation of comparable benchmarks between these implementations. To get a better understanding of the results, the chapter starts with section 3.1 describing the tools and methodology used for benchmarking. The following implementations details of section 3.2 provide precise internals of the benchmarking suite. This might be used for further development of the software. A description of how to actually use the presented benchmarking suite will be given in section 3.3. Finally, the benchmarking results are listed in **??**.

3.1. Benchmarking Methodology

In order to generate independent and stable benchmarking results, the benchmarking suite runs within a virtual environment: *Docker* is used to separate the running suite from the host operating system. This reduces the influence of resource intensive processes, which might falsify benchmarking results. Moreover, *Docker* enables a portable and scalable software solution.
In the following, the process of benchmarking is described within five steps. In a short version, the required steps are:

1. Create the benchmarking source code
2. Compile the benchmarking source code
3. Run *Callgrind*
4. Run *Massif*
5. Collect benchmarks

Create the benchmarking code

Each software libraries presented in chapter 2 implements various cryptographic primitives. To avoid overhead when calculating benchmarks, only the required functions to generate a SIDH key exchange should be called. Thus, a simple benchmarking file is provided for each implementation, which calls the appropriate underlying API. This benchmarking file must ensure that all required headers are imported, the API is called correctly and a `main`-function is provided.

Compile the benchmarking code

Once the benchmarking source code is created, it needs to be compiled to a binary. Therefore, all required dependencies (libraries, headers, sourc code, ..) need to be provided to the compiler. The benchmarking suite provides a `Makefile` for each implementation, where the compilation process is implemented. All compilations performed for the benchmarking suit must use consistent compiler optimizations. This ensures the comparability of the outputs. Currently, the optimization flag `-O3` is passed to the gcc compiler. Note, that CIRCL is implemented in GO and therefore the go compiler is used for compilation. Since this compiler does not provide optimization flags, the default compiler optimizations are used [35]. To be able to extract benchmarks for specific functions *inlining* is disabled during compilation. The C programming language offers the following directive to disable *inlining* for a function:

```c
// This function will not be inlined by the compiler
void __attribute__ ((noinline)) no_inlining() {
// ...
}

// This function might be inlined by the compiler
void inlining() {
// ...
}
```

The go compiler can be directly invoked with a no-inlining (`-l`) flag:

```
go build -gcflags "-l"
```

Run *Callgrind*

Callgrind records function calls of a binary. For each call the executed instructions are counted. Moreover, the tool provides detailed information about the callee and how often functions are called.Thus, the tool also provides information about execution hotspots of the binary. *Callgrind* is part of *Valgrind*, a profiling tool which allows deep analysis of executed binaries. Callgrind is invoked on the command line via:

```
valgrind --tool=callgrind --callgrind-out-file=callgrind.out binary
```

The profiling data of callgrind is written to the file defined by `-callgrind-out-file`. This file might be analyzed using a graphical tool like *KCachegrind* or any other analyzing script. Running a binary using callgrind, slows down the execution times significantly. This is the main reason for the long execution times of the benchmarking suite.

Run *Massif*

Massif measures memory usage of a binary, including heap and stack. *Massif* is also part of the profiling tool *Valgrind*. The tool creates multiple snapshots of the memory consumption during execution. Thus, one can extract the maximum memory consumption of a binary. The following command runs the *Massif*:

```
valgrind --tool=massif --stacks=yes --massif-out-file=massif.out binary
```

The profiling data will be written to the file defined by `-massif-out-file`. *Massif-visualizer* could be used to graphically analyze the data.

Collect benchmarks

Once the output files of *Callgrind* and *Massif* are produced, one can analyze the corresponding files to obtain:

1. Absolute instructions per function.
2. Maximum memory consumption during SIDH key exchange.

This information is finally used by the benchmarking suite, to produce graphs and tables for further investigation. To receive reliable information, all benchmarks are executed multiple times.
To further increase the quality and reproducibility of the results, the cache of the virtual operating system is cleared, before *Callgrind* and *Massif* are executed. This is done via:

```
sync; sudo sh -c "echo 1 > /proc/sys/vm/drop_caches"
sync; sudo sh -c "echo 2 > /proc/sys/vm/drop_caches"
sync; sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"
```

3.2. Application Details

The benchmarking suite is developed in Python3 on a Linux/Ubuntu operating system. Currently, the following implementations are included:

- SIKE working on: `p434, p503, p610, p751`
  - Sike reference implementation (`SIKE_Reference`)
  - Sike generic optimized implementation (`SIKE_Generic`)
  - Sike generic optimized and compressed implementation (`SIKE_Generic_Compressed`)
  - Sike x64 optimized implementation (`SIKE_x64`)
  - Sike x64 optimized and compressed implementation (`SIKE_x64_Compressed`)
- PQCrypto-SIDH working on: `p434, p503, p610, p751`
  - PQCrypto-SIDH generic optimized implementation (`Microsoft_Generic`)
  - PQCrypto-SIDH generic optimized and compressed implementation (`Microsoft_Generic_Compressed`)
  - PQCrypto-SIDH x64 optimized implementation (`Microsoft_x64`)

  - PQCrypto-SIDH x64 optimized and compressed implementation (`Microsoft_x64_Compressed`)
- CIRCL working on: `p434, p503, p610, p751`
  - CIRCL x64 optimized implementation (`CIRCL_x64`)

Furthermore, the benchmarking suite provides classical elliptic curve Diffie-Hellman (ECDH) based on OpenSSL. Since SIDH is a candidate to replace current Diffie-Hellman algorithms, ECDH is intended as reference value: The objective is it to compare optimized modern ECDH with quantum-secure SIDH.
Since each parameter set of SIDH meet a different security level (compare subsection 1.3.4) ECDH is also instantiated with different curves, each matching a SIDH security level. The used elliptic curves are namely:

1. `secp256r1` (openssl: `prime256v1` [36]): 128 bit security matching `SIKEp434` [37]
2. `secp384r1` (openssl: `secp384r1`): 192 bit security matching `SIKEp610` [37]
3. `secp521r1` (openssl: `secp521r1`): 256 bit security matching `SIKEp751` [37]

For each of these introduced implementations the application measures benchmarks. In the following sections detailed information about the internals of the suite is given. Beside the precise application flow (subsection 3.2.1), the internal class structure of the Python3 code is shown (subsection 3.2.2).

3.2.1. Application Flow

This sections illustrates the application flow of the benchmarking suite (see Figure 3.1). Once triggered to run benchmarks, the following procedure is repeated for every implementation: The suite first compiles the benchmarking code. The binary is then executed multiple times to generate N benchmarking results, respectively for *Callgrind* and *Massif*.
Finally, the results are visualized in different formats. All resulting values are averages over N samples. Graphs compare the recorded instruction counts and the peak memory consumption among implementations instantiated with comparable security classes. The HTML and Latex tables lists all benchmarks per implementation and shows the standard derivation over N samples.

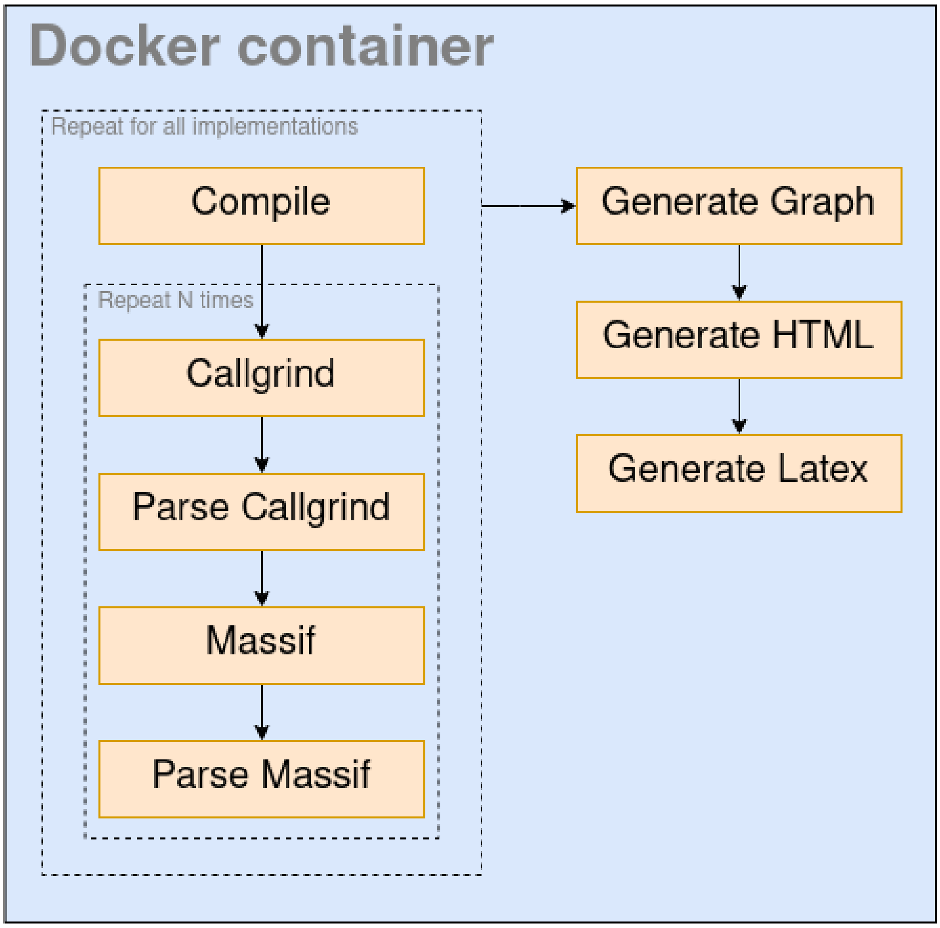

Figure 3.1.: Flow chart of the benchmarking suite. For each SIDH implementation, the source code is compiled and benchmarked multiple times. The benchmarking results are visualized in different format: Graphs, HTML and Latex.

3.2.2. Application Structure

This section covers the internal structure of the benchmarking suite. It illustrates, how each implementation is represented in code and how the benchmarking results are managed. To get a detailed description of the implemented functions have a look into the well-documented source code.

Representation of concrete implementations

The main logic of the benchmarking suite is placed within the class `BaseImplementation`. This class implements the logic of compiling code and measuring benchmarks. For each implementation which shall be benchmarked, a subclass of `BaseImplementation` is created (see Figure 3.2). These subclasses provide a link to the respective *Makefile*, which is used by the `BaseImplementation` to compile code, run callgrind and run massif. Furthermore, each subclass can provide a set of arguments passed to the *Makefile*.

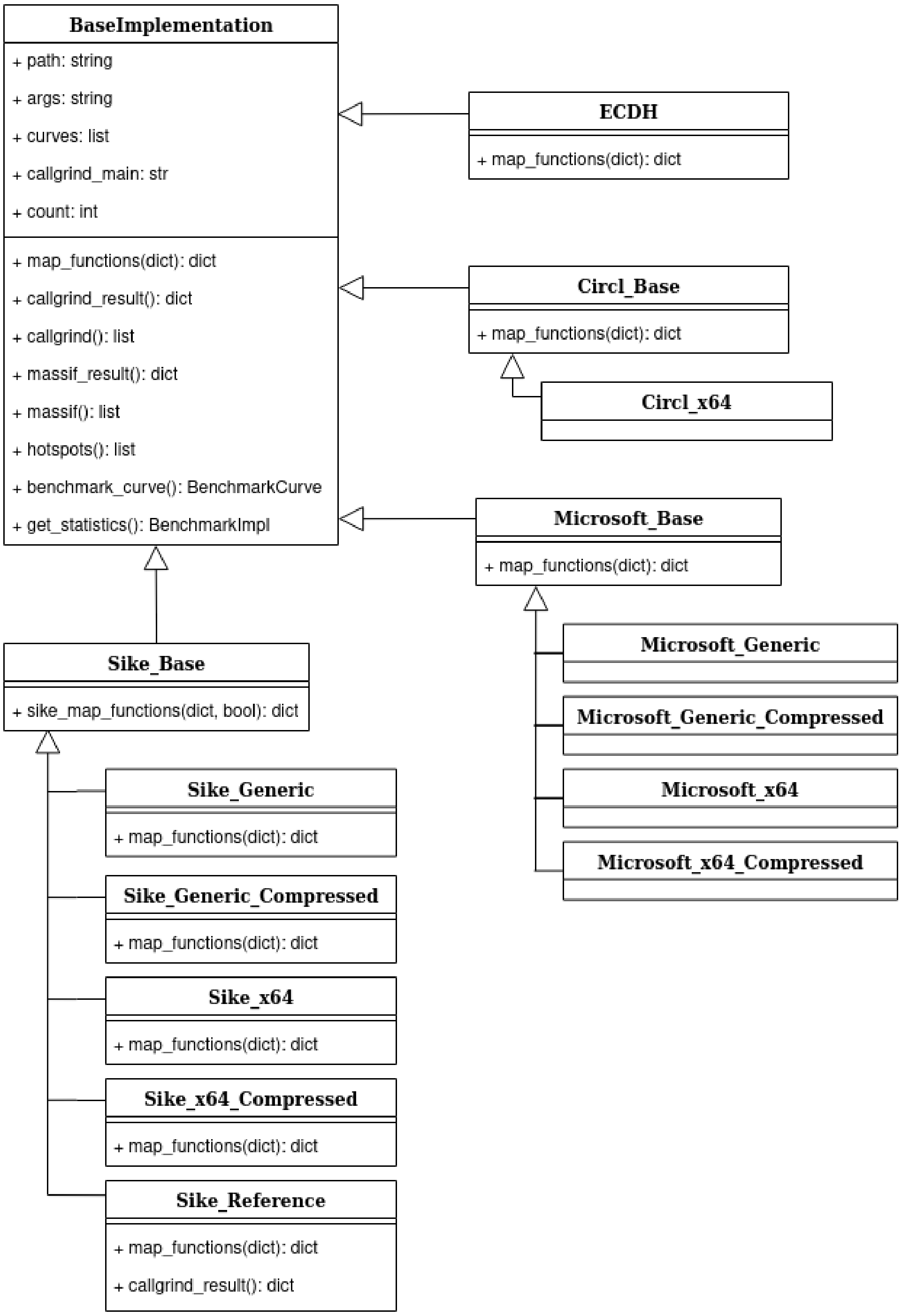

Figure 3.2.: Class diagram for the all implementations supported by the benchmarking suite. All concrete implementations of SIDH inherit from `BaseImplementation`. A *Makefile* provided by each subclass specifies how the benchmarking code is built and run.

Representation of benchmarking results

In order to ensure a clear analysis of the results the application implements a structure for the returned benchmarking values. Since each implementation can be run based on different parameter sets (in the terminology of the benchmarking suite this is called *curves*) and for each curve different benchmarking values are processed, the following hierarchy is applied (see Figure 3.3).
The class `BechmarkImpl` represents the benchmarking results of a specific implementation. Therefore, the class manages a list of `BenchmarkCurve` objects, which contains benchmarks for a specific curve. Each benchmark for such a curve is represented by an instance of `Benchmark`. Thus, `BenchmarkCurve` holds a list of `Benchmark` objects.

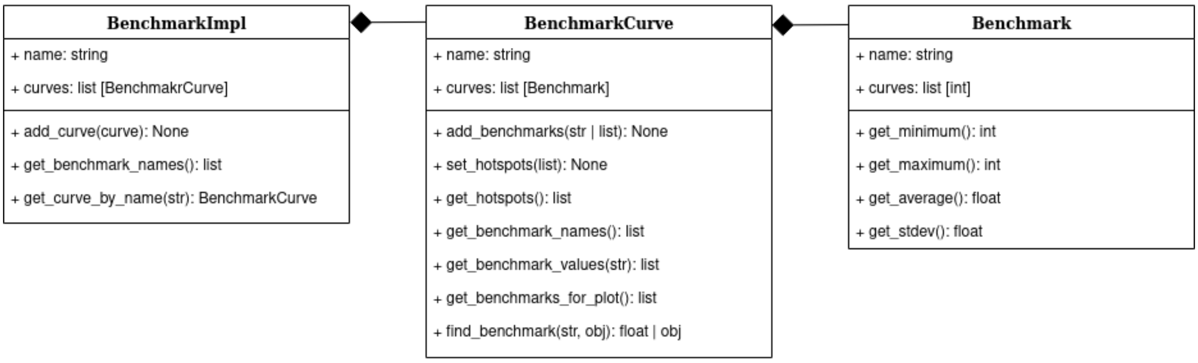

Figure 3.3.: Class diagram representing the management of the benchmarking results.

3.2.3. Adding Implementations

To add a new implementation to the benchmarking suite it is helpful the have a look to existing implementations. Adding a SIDH implementation to the benchmarking suite requires the following steps:

1. Add a new folder into the `container/` folder of the root directory `container/new_implementation/`.
2. Create a file `container/new_implementation/benchmark.c` that calls the SIDH key exchange API of the new implementation.
3. Add all necessary dependencies into `container/new_implementation/` that are needed to compile `benchmark.c`. Note, maybe some changes in the *Dockerfile* are necessary to install certain dependencies on the virtual operating system started by Docker.
4. Create a `Makefile`, that supports the following commands:
   - build: Compile `benchmark.c` into the executable `new_implementation/build/benchmark`
   - callgrind: Runs callgrind with the executable and stores the result in `new_implementation/benchmark/callgrind.out`.

- massif: Runs massif with the executable and stores the result in `new_implementation/benchmark/massif.out`.

5. Add a new class to the source code, which inherits from `BaseImplementation`. You need to overwrite the `map_functions()` method to map the specific API functions of the new implementation to the naming used within the benchmarking suite. The new class might look similar to this:

```python
class New_Implementation(BaseImplementation):
    def __init__(self, count):
        super().__init__(count=count,
                                path=[path to Makefile],
                                args=[args for Makefile],
                                callgrind_main=[name of main function],
                                curves=curves)

    def map_functions(self, callgrind_result: dict) -> dict:
        res = {
            "PrivateKeyA": callgrind_result[[name of api function]],
            "PublicKeyA": callgrind_result[[name of api function]],
            "PrivateKeyB": callgrind_result[[name of api function]],
            "PublicKeyB": callgrind_result[[name of api function]],
            "SecretA": callgrind_result[[name of api function]],
            "SecretB": callgrind_result[[name of api function]],
        }
        return res
```

6. Import the created class into `container/benchmarking.py` and add the class to the `implementations` list:

```python
implementations =[
        #...
        New_Implementation,
    ]
```

Once these steps are done the benchmarking suits is able to benchmark the new implementation.

3.3. Usage

This section explains the usage of the benchmarking suite. Beside the execution of benchmarks, the application also provides unit tests for the implementation. Both tasks, running benchmarks and executing unit tests, need to run within the docker container. Thus, it is mandatory to install docker on your system to run the benchmarking suite[1] To provide a easy to use interface for both tasks a script *run.sh* is available. This script can be run with different arguments:

- Building the docker container:

  ```
  ./run.sh build
  ```

- Running unit tests within the docker container.

  ```
  ./run.sh test
  ```

  Testing the benchmarking suite runs for a while, since each implementation is compiled to verify the functionality of the *Makefiles*. However, this procedure is logged while the unit tests are executed.

- Running benchmarks within the docker container.

  ```
  ./run.sh benchmark
  ```

  This command triggers the *benchmarking.py* script within the docker container. This script contains the entry logic of the benchmarking suite. It benchmarks all available implementations and generates different output formats (graphs, html, latex). It is configurable how often each benchmark should be measured: The variable `N` in `container/benchmarking.py` describes the amount of repetitions for *callgrind* and *massif*. Once the benchmarks are done, all output files can be inspected in the folder `data/` of your current directory. These files visualize average values over N samples. The output files are:

  - *XXX.png*: These files compare the absolute instruction count of all implementations, which were run on the XXX parameter set (XXX $\in 434, 503, 610, 751$). Additionally, it contains a ECDH benchmark for comparison.
  - *XXX_mem.png*: These files compare the peak memory consumption of all implementations, which were run on the XXX parameter set (XXX $\in 434, 503, 610, 751$). Additionally, it contains a ECDH benchmark for comparison.
  - *cached.json*: This cache file contains all benchmarking results. It can be used as input for the benchmarking suite to use cached data instead of generating all benchmarks again. Since the benchmarking suite runs multiple hours if each implementation is evaluated N=100 times, this functionality provides great speed up if only the output data should change. To use the cached file as input, copy it to `container/.cached/cached.json` and run the benchmarking suite again.
  - *results.html*: This file lists detailed benchmarking results in a human readable HTML table.

[1] Instructions for installing *Docker*: https://docs.docker.com/get-docker/
(The application was developed using Docker version 19.03.8, build `afacb8b7f0`)

– *results.tex*: This file contains the benchmarking results formatted in a latex table. The output is used for this document.

4. Benchmarking Results

The results presented in this chapter were calculated on a x64 architecture (Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz) running Ubuntu 20.04.1 LTS. The installed docker version was 19.03.8. The benchmarking suite was initialized to run callgrind and massif 100 times, respectively (e.g. N=100).

The exact values measured (averages and standard derivation over N=100 samples) and the identified execution hotspots for all implementations instantiated respectively with all parameter sets (see section 3.2) can be found in addenda A.1. The efficiency in this chapter will be quantified by the peak memory consumption and by the absolute instruction count measured by the benchmarking suite (details in chapter 3).

The graphs presented in this chapter use the following terminology: Assume Alice (A) and Bob (B) want to establish a shared secret using a SIDH key exchange.

- *KeygenA* describes the key generation (public and private key) of Alice.
- *KeygenB* describes the key generation (public and private key) of Bob.
- *SecretA* describes the computation of the shared secret by Alice.
- *SecretB* describes the computation of the shared secret by Bob.

This chapter starts with a comparison between all SIDH security levels in section 4.1. This demonstrates the performance differences of the available parameter sets.

A comparison among all presented SIDH libraries is given in section 4.2: Firstly, 4.2.1 works out differences between *compressed* and *optimized* SIDH implementations. Then, all optimized variants (*generic* and *x64*) are compared in 4.2.2. The compressed versions of all libraries are put into relation in 4.2.3. Finally, a overview about the measured execution hotspots for all libraries is given in 4.2.4.

Differences in terms of efficiency between modern state-of-the-art ECDH and quantum-resistant SIDH are pointed out in section 4.3. This section also highlights execution hotspots measured for ECDH.

The chapter ends with security considerations for all benchmarked implementations in terms of constant time cryptography and key size (section 4.4).

4.1. Comparing SIDH security levels

Before any differences between libraries or implementations are drawn this section considers the different parameter sets proposed in [24]: `p434`, `p503`, `p610` and `p751`. All these parameters match a security level defined by NIST (see subsection 1.3.4 for details). This section makes use of the `SIKE_x64` implementation to visualize the claimed resources of all parameter sets. However, the chosen implementation does not effect the results for this comparison .

Figure 4.1 shows the absolute instruction counts for `SIKE_x64` initialized with all available parameter sets. While `p434` executes 18 million instructions for *KeygenA*, `p751` requires 67 million operations (3.7 times more). Roughly the same can be seen for *KeygenB*, *SecretA* and *SecretB*. The other parameter sets `p503` and `p610` almost lie on a linear line between the highest and lowest security class.

Similarly, the measured peak memory consumption is for `p751` the highest (13.3 kB) and for `p434` the lowest (8.2 kB). Again, the other parameter sets can be found in between of these boundary values (Figure 4.2).

This analysis clearly shows that a increased security level for SIDH corresponds with a increased claim of resources. While the difference in terms of memory consumption is relatively small the execution times differ a lot: On average and compared to `p434` the parameter

- `p503` executes 1.4 times more instructions
- `p610` executes 2.5 times more instructions
- `p710` executes 3.7 times more instructions

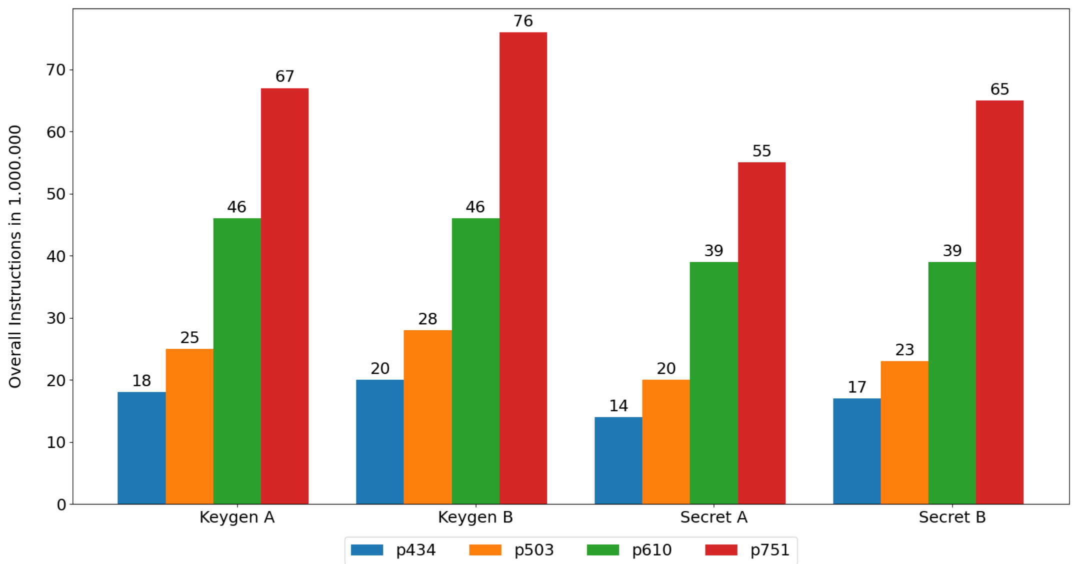

Figure 4.1.: Overall instructions for `SIKE_x64` initiated with all possible parameter sets.

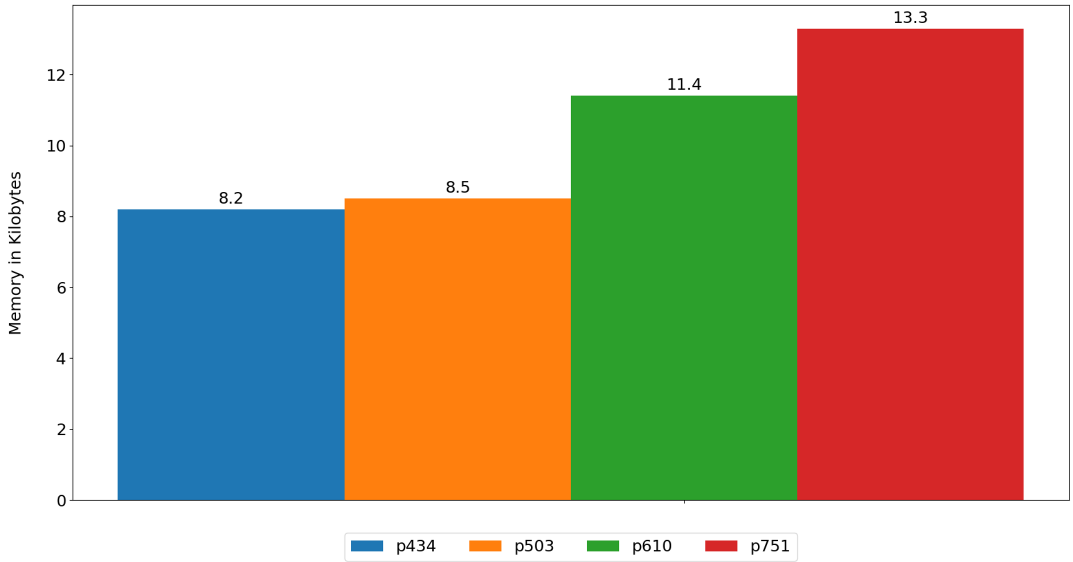

Figure 4.2.: Maximum memory consumption in kilobytes of `SIKE_x64` initiated with all possible parameter sets.

4.2. Comparing SIDH libraries

This section compares the following SIDH libraries: *SIKE*, *PQCrypto-SIDH* and *CIRCL*. For a detailed description of these libraries see chapter 2.

4.2.1. Comparing SIKE Implementations

Before comparisons between the SIDH libraries *SIKE*, *PQ-Crypto* and *CIRCL* are drawn, the performance differences between *optimized* and *compressed* versions are investigated. Exemplarily, all variants of SIKE are taken into considerations. Note that all *compressed* versions of SIKE also contain *optimized* code. Their key sizes, however, are reduced compared to non-compressed variants.
Figure 4.3 clearly shows a difference between *optimized* and *compressed* versions of SIKE in term of absolute instructions. Compressed versions roughly claim twice as much operations (~500 million for `SIKE_Generic_Compressed` and ~45 million `SIKE_x64_Compressed`) to generate a key pair as non-compressed variants (~200 million for `SIKE_Generic` and ~19 million `SIKE_x64`). Additionally, the generation of the shared secret is a little slower for the compressed implementations.
Besides execution times, Figure 4.4 also compares the memory consumption. As stated by the authors of SIKE, compressed variants allocate more memory: The peak memory allocation is with 17 kilobytes (`SIKE_Generic_Compressed`) and 19.2 kilobytes (`SIKE_x64_Compressed`) twice as high as the non-compressed versions.
The `SIKE_Reference` implementation is with 11.2 kB allocated memory peak close to the

average of all SIKE implementations. However, regarding the execution times for key generation, the reference implementation is by far the slowest variant: The more than 2.3 billion (2.300.000.000) instructions were measured in order to generate Bobs key pair. This is slower by factor 100 compared to `SIKE_x64`. As the name suggests, this reference implementation must be seen as a proof-of-concept. Thus, it will not be considered further in this analysis.

To sum up, the comparison between all SIKE implementations showed, that *optimized* versions have decreased instructions counts as well as decreased memory consumptions compared to *compressed* implementations. The further analysis of the detailed benchmarks in in addenda A.1 lead to a very similar result for the *PQ-Crypto* library of Microsoft. One can state, that *compressed* versions roughly demand twice as much resources than *optimized* variants.

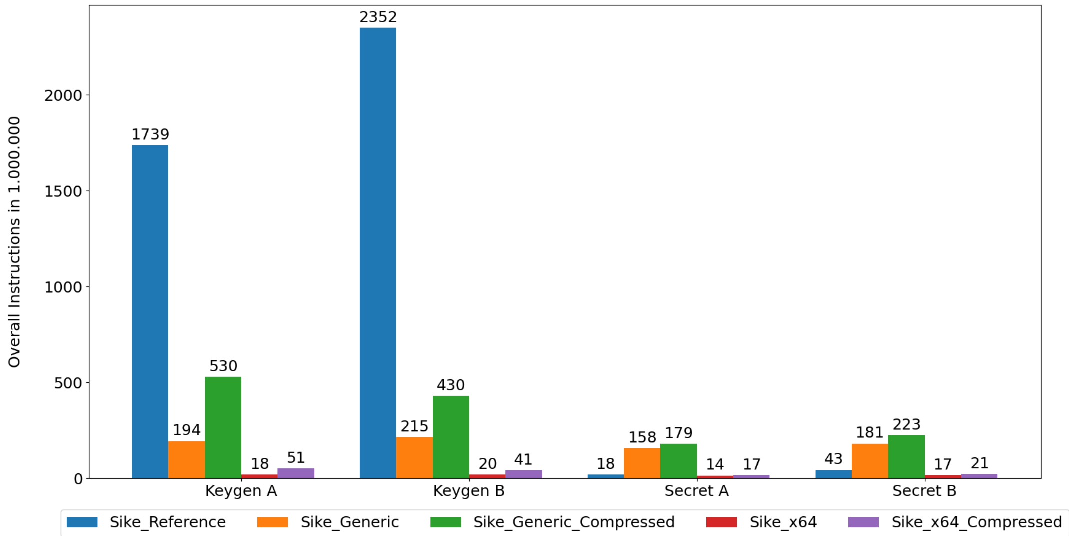

Figure 4.3.: Overall instructions for all SIKE implementations initialized with `p434`. The reference implementation is the slowest, the x64 optimized version is the fastest. These results meet intuitive expectations.

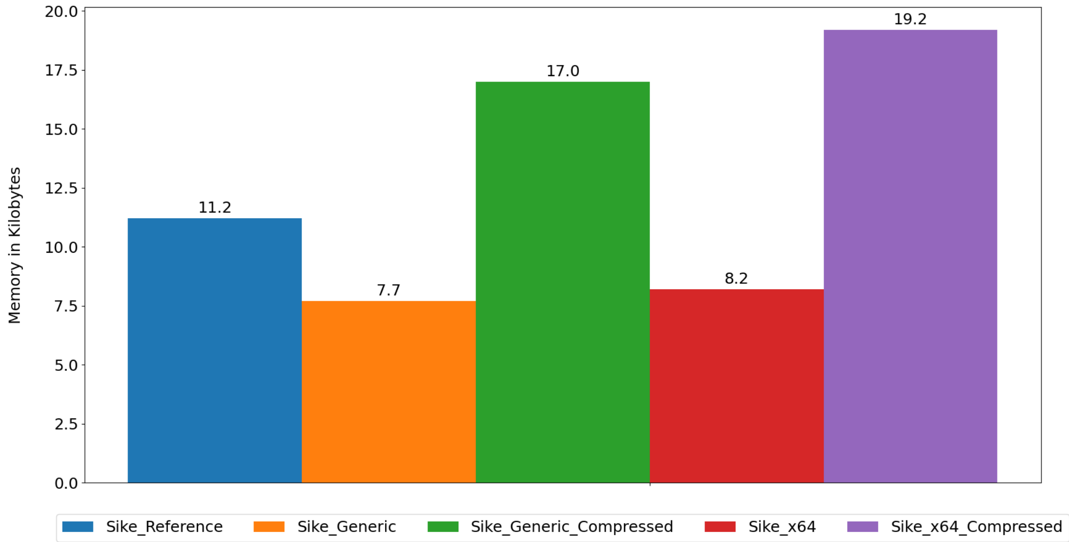

Figure 4.4.: Maximum memory consumption in kilobytes or all SIKE implementations initialized with p434. The required memory overhead of compressed versions is clearly visible.

4.2.2. Comparing optimized implementations

Generic optimized implementations

Generic optimized implementations contain generally optimized code for various hardware platforms. However, no hardware specific instructions can be exploited in these versions. The benchmarking suite currently supports [1]:

1. *SIKE_Generic*
2. *Microsoft_Generic*

Figure 4.5 shows the benchmarks for execution time of all optimized variants initiated with p434. For this section, however, only *SIKE_Generic* and *Microsoft_Generic* are taken into account. In all four steps of the SIDH key exchange the benchmarked values for both implementations look almost the same. However, when considering exact measurements from A.1, one can observe that *SIKE_Generic* executes constantly half a million operations less than *Microsoft_Generic* (this holds for all four categories: Keygen A, Keygen B, Secret A and Secret B). While this sounds significant, the relative difference is actually less than 0.01%. While the absolute gap further increases if higher security classes are analyzed (about three million operations constant difference for p751), the relative disparity stays smaller than 0.01%.

Peak memory consumptions for parameter *p434* are visualized in Figure 4.6. As in terms of execution time, memory allocation numbers between *SIKE_Generic* and *Microsoft_Generic*

[1] Although CIRCL states to offer a generic optimized implementation[33], this version could not be compiled.

hardly differ: The SIKE version occupies 0.3 kB less memory for *p434* and 0.7 kB less than for *p751*. Overall, the relative difference regarding memory consumption is about 5%.

Although both implementations hardly differ in their performance benchmarks one can state, that *SIKE_Generic* demands less resources than *Microsoft_Generic*. This disparity, however, is marginal.

x64 optimized implementations

X64 optimized implementations exploit AMD64 specific hardware operations to improve performance on these machines. The benchmarking suite currently supports the following x64 optimized variants:

1. *SIKE_x64*
2. *Microsoft_x64*
3. *CIRCL_x64*

Figure 4.5 shows the execution time benchmarks for all optimized variants initiated with `p434`. For this section, however, only *SIKE_x64*, *Microsoft_x64* and *CIRCL_x64* are taken into account. In all four categories listed in the graph *Microsoft_x64* is the fastest implementation. *SIKE_x64* is slightly slower executing about two million instructions more in each category. This corresponds to a relative difference of about 10%. The most expensive implementation in terms of performed operations is *CIRCL_x64*: 40% more operations are needed in each step of the SIDH key exchange compared to the fastest variant *Microsoft_x64*.
Figure 4.7 compares the implementations initiated with *p751* - matching the highest NIST security level 5. *Microsoft_x64* stays the fastest version while the realtive difference to *SIKE_x64* (5%) and *CIRCL_x64* (30%) decreases.

To compare the memory consumption of the x64 optimized implementations, have a look at Figure 4.6. The memory benchmarks of *SIKE_x64* (8.2 kB) and *Microsoft_x64* (8.9 kB) hardly differ, whereas *CIRCL_x64* has a peak allocation of 24.7 kB for a single SIDH key exchange. This is by factor 2.5 greater compared to the others. However, Figure 4.8 reveals that the memory consumption for *CIRCL_x64* does barely change for higher security classes. Nevertheless, *CIRCL_x64* has the most intense memory consumption of all x64 optimized implementations and *Microsoft_x64* allocates roughly about 10% more memory than *SIKE_x64* (this holds respectively for all security classes) .

The fastest x64 optimized SIDH key exchange is performed by the Microsoft library *PQ-Crypto*. *SIKE_x64* is slightly slower but allocates less memory than *Microsoft_x64*. The most resources are consumed by *CIRCL_x64* (instruction count and memory allocations).

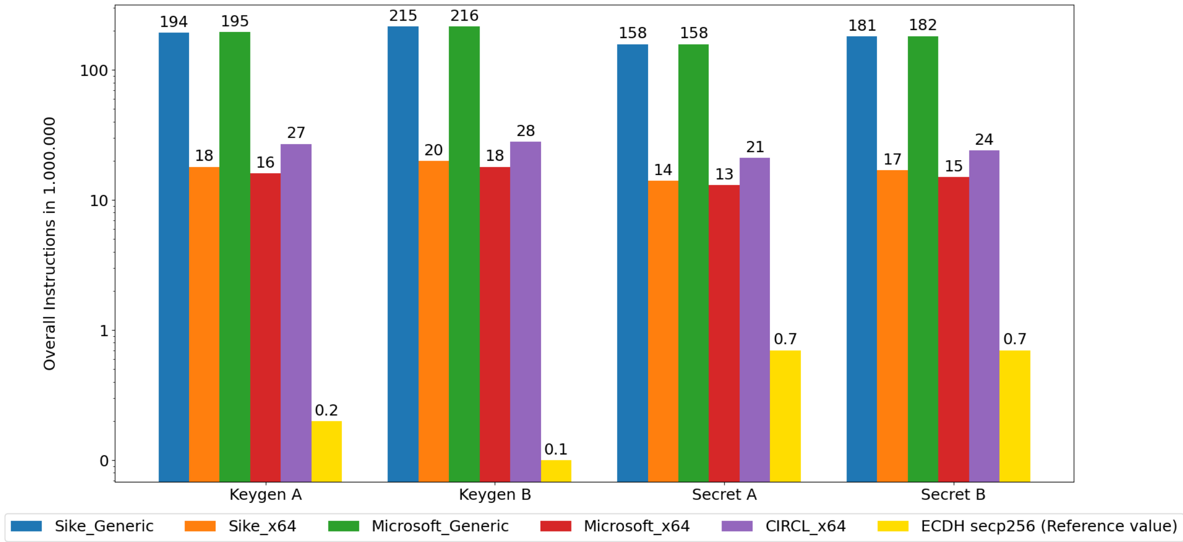

Figure 4.5.: Overall instructions for SIDH parameter `p434` compared to ECDH via `secp256r1`.

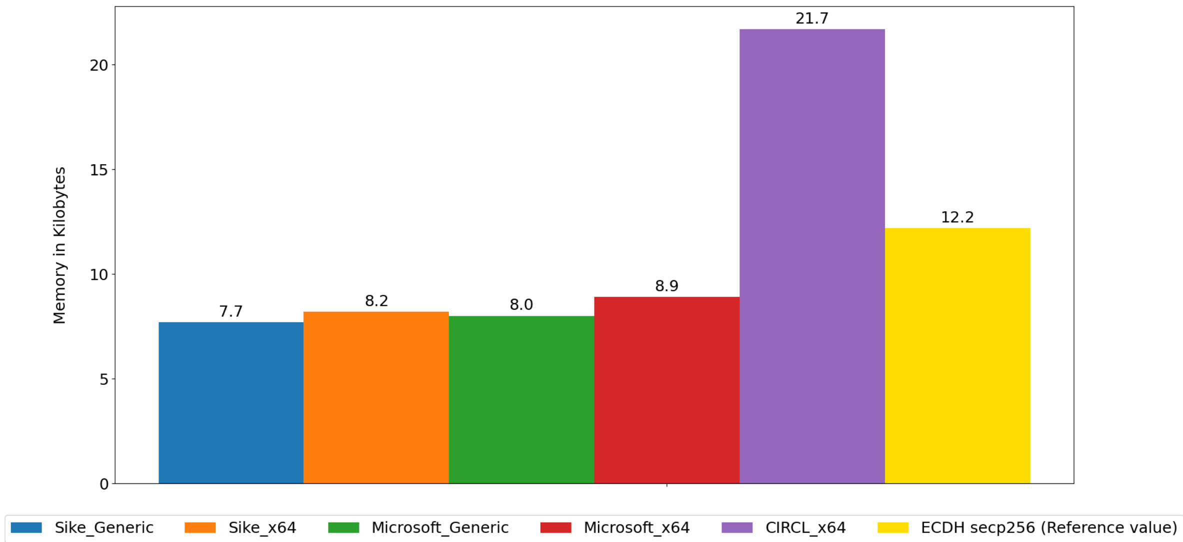

Figure 4.6.: Maximum memory consumption in kilobytes for SIDH parameter `p434` compared to ECDH via `secp256r1`.

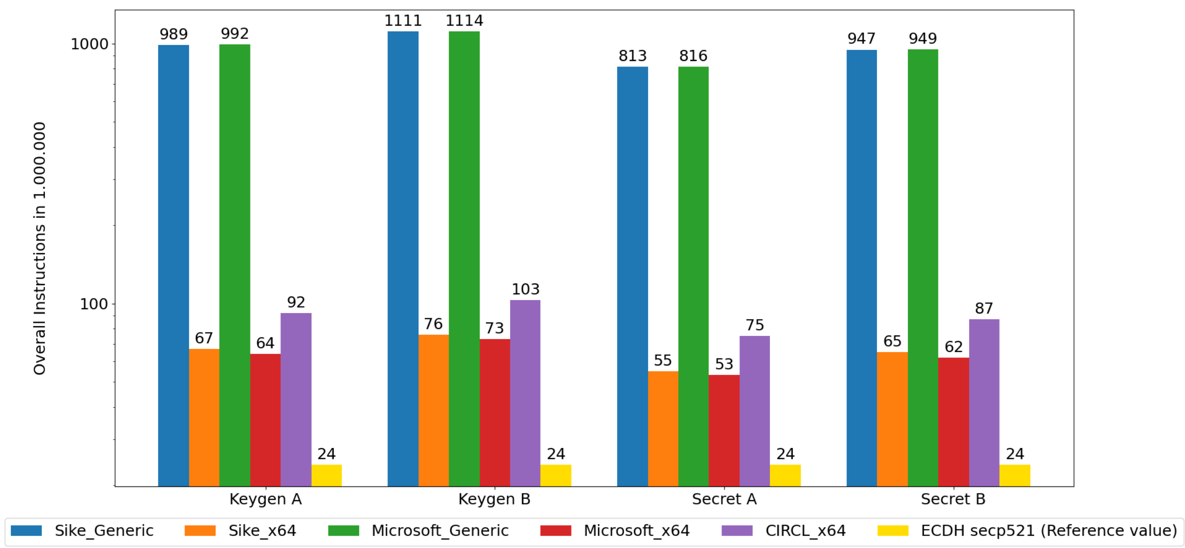

Figure 4.7.: Overall instructions for SIDH parameter `p751` compared to ECDH via `secp521r1`.

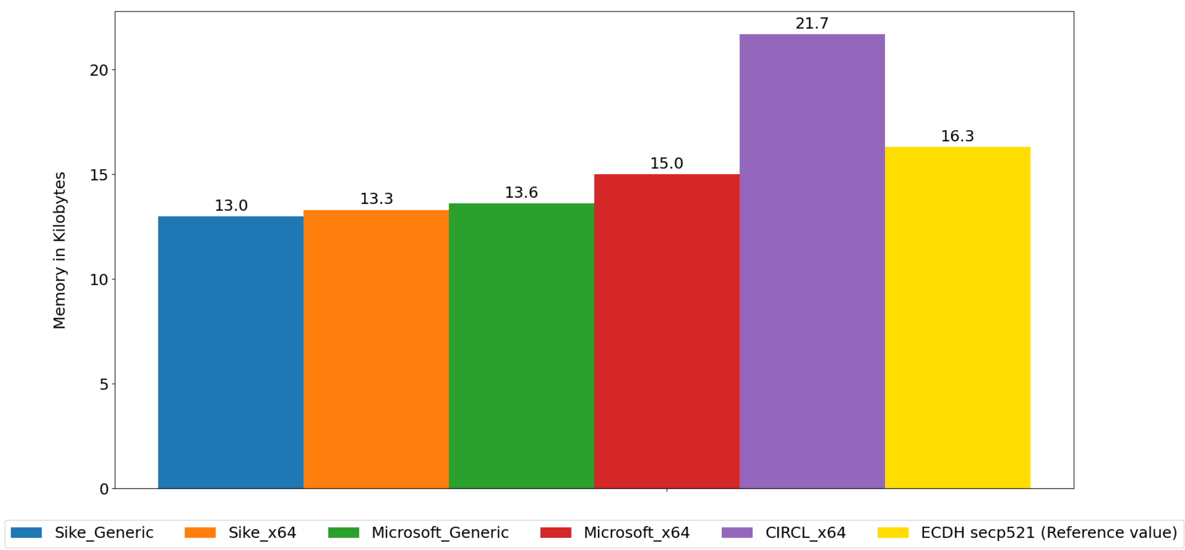

Figure 4.8.: Maximum memory consumption in kilobytes for SIDH parameter `p751` compared to ECDH via `secp521r1`.

4.2.3. Comparing compressed implementations

Compressed implementations of SIDH promise shortened key size compared to non-compressed versions, however, this leads to increased execution times and memory allocations. The benchmarking suite currently support the following compressed versions:

1. *SIKE_Generic_Compressed*

2. *SIKE_x64_Compressed*
3. *Microsoft_Generic_Compressed*
4. *Microsoft_x64_Compressed*

Figure 4.9 shows the execution time benchmarks for all compressed variants initiated with p434. Naturally, the x64 optimized code is faster than the generic optimization. The Microsoft implementations performed less operations than SIKE: Regarding key generation, *SIKE_Generic_Compressed* performed on average 38% more instructions than *Microsoft_Generic_Compressed* and *SIKE_x64_Compressed* performed on average 45% more instructions than *Microsoft_x64_Compressed*. The generation of the secret key only shows small differences between SIKE and Microsoft, however Microsoft is still the faster implementation. The trend of this analysis also applies for improved security classes, e.g for parameter *p751* (see Figure 4.11)

The following evaluation of the allocated memory is surprising (Figure 4.10): The Microsoft implementations occupy three times more memory than SIKE when initiated with *p434*. This gap rises strongly when increasing the security class to *p751* (Figure 4.12), where the the *PQ-Crypto* library of Microsoft allocates almost seven times more memory.

While the benchmarks for the compressed Microsoft implementations show faster execution times, the overhead of allocated memory compared to SIKE is enormous.

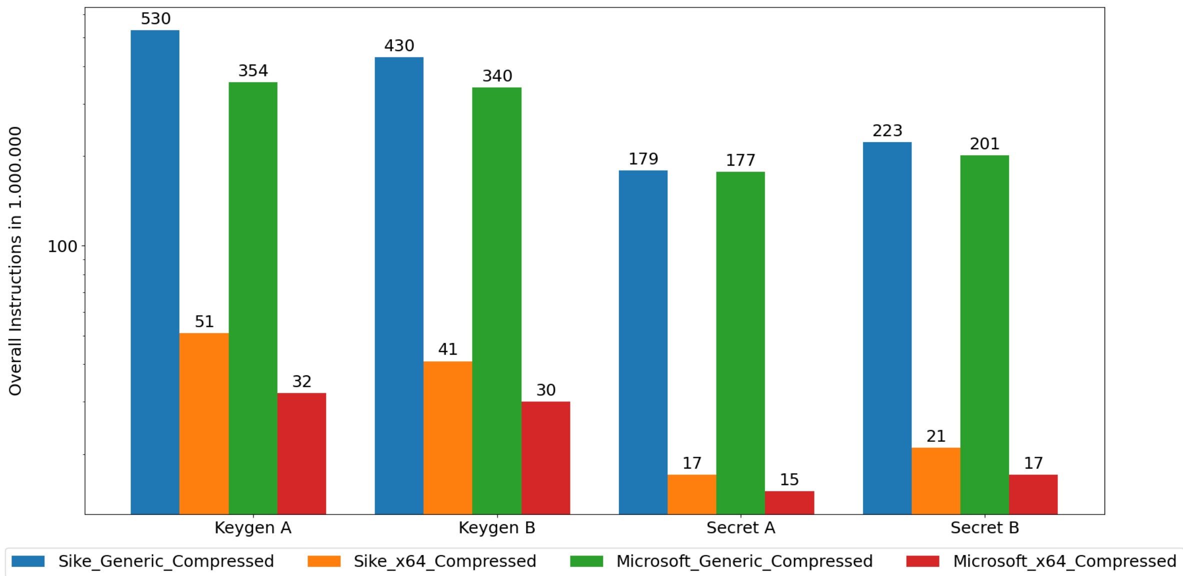

Figure 4.9.: Overall instructions for compressed SIDH parameter p434.

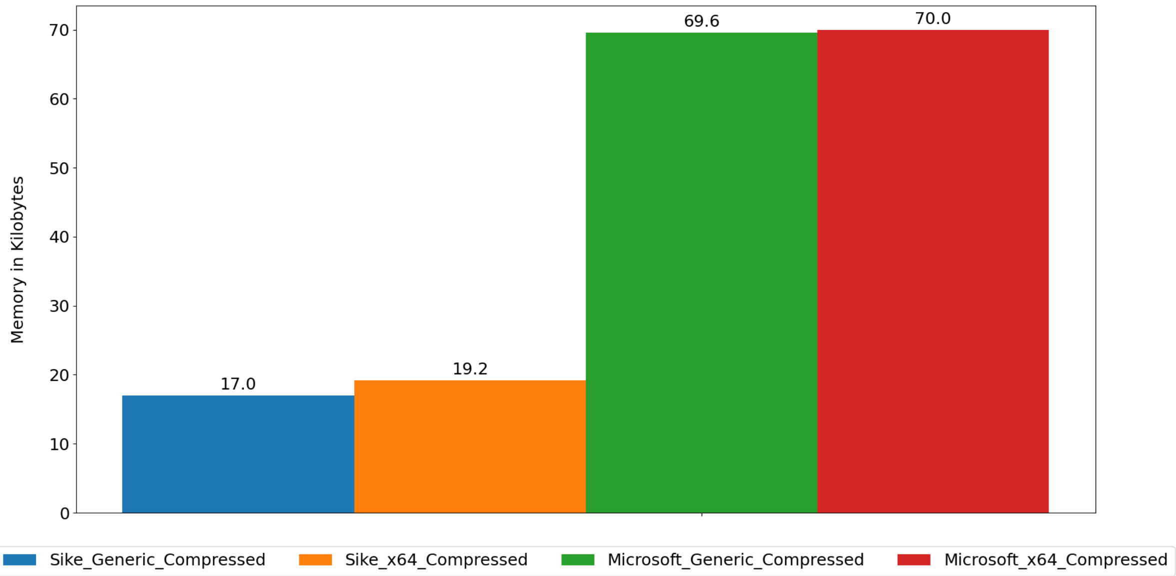

Figure 4.10.: Maximum memory consumption in kilobytes for compressed SIDH parameter p434.

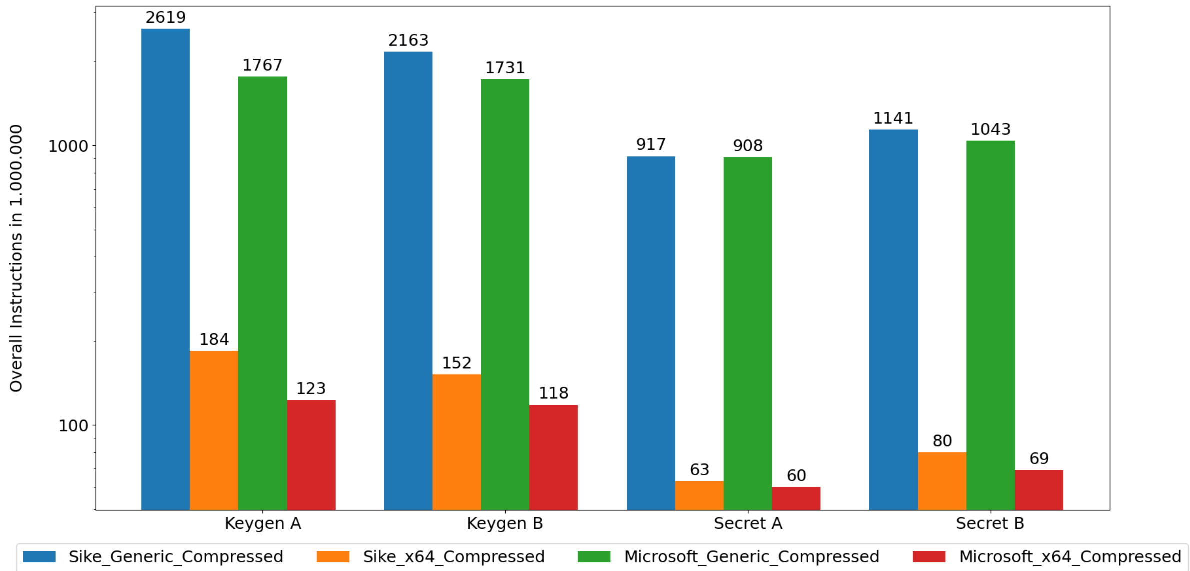

Figure 4.11.: Overall instructions for compressed SIDH parameter p751.

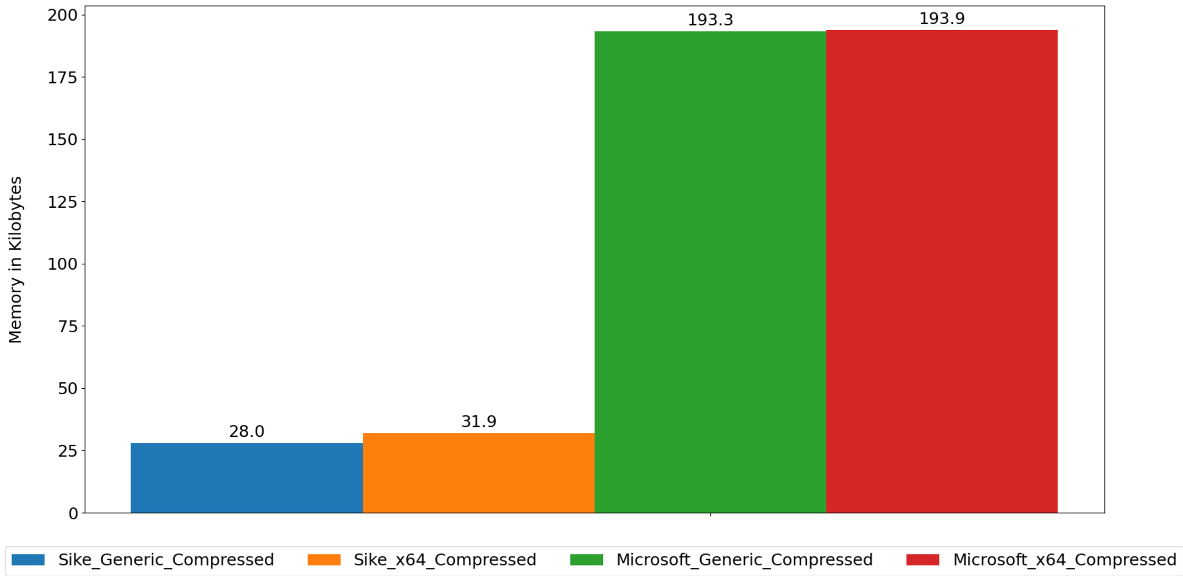

Figure 4.12.: Maximum memory consumption in kilobytes for compressed SIDH parameter `p751`.

4.2.4. Analysis of execution hotspots

Beside the detailed benchmarks addenda A.1 also lists the execution hotspots of each implementation initialized with different parameters. The following description of these hotspots reveals potential methods where performance of SIDH can be further improved.
This evaluation also shows the similarity of *SIKE* and *PQ-Crypto* which apparently build their SIDH APIs upon the same source code. Thus, both libraries suffer from the same execution hotspots: Each variant of *SIKE* and *PQ-Crypto* spends more than 50% of its execution time within the function `mp_mul`. The second great hotspot of both libraries is the function `rdc_mont` with up to 32% consumed operations:

1. `mp_mul` calculates $c = a * b$ for two given n-digit integers a and b (based on Karatsubas multiplication algorithm).

2. `rdc_mont` calculates $c = a \ mod \ p$ for given integers a and p (based on Montgomery reduction).

The identified hotspots of *CIRCL* are namely `mulPxxx` (~50%) and `rdcPxxx` (~50%) where `xxx` $\in \{434, 503, 751\}$. The source code provides further information, however, the documentation of *CIRCL* makes it hard to do reliable statements on the exact internals of the identified hotspot functions:

1. `mulPxxx` calculates $z = x * y$ given integers a and b (based on Karatsubas multiplication algorithm).

2. `rdcPxxx` calculates $z = x * R^{-1}(mod 2 * p)$ for given integers x and p (based on Montgomery reduction).

It can be seen that all three SIDH libraries struggle with the same issue: Performing many multiplications and modulo operations is expensive. Since all libraries exploit state-of-the-art algorithms (Karatsubas multiplication algorithm and Montgomery reduction) the ongoing research in supersingular isogeny cryptography needs to find other ways to improve performance. Especially when comparing SIDH with modern ECDH key exchanges, these limitations are clearly visible.

4.3. Comparing SIDH and ECDH

To be able to make reliable statements about the current state of SIDH this section compares the quantum-secure SIDH implementations with state-of-the-art elliptic curve Diffie-Hellman (ECDH).
Beside all optimized SIDH versions Figure 4.5 also shows benchmarks for a ECDH reference value via `secp256r1`. Note, that the security class of parameter set `p434` matches with `secp256r1` (see section 3.2 for details). Compared to the fastest SIDH optimized implementation (`Microsoft_x64`) ECDH is significantly faster: 80 times less instructions for KeygenA and 180 times less instruction for KeygenB are executed. Additionally, the generation of the secret is 18 times faster for secretA and 21 times faster for secretB.
More moderate results can be observed for parameter set `P751` and `secp512` (Figure 4.7): KeygenA (2.5 times), KeygenB (3 times), SecretA (2.2 times) and SecretB (2.6 times) of `secp512` are faster compared to the fastest SIDH variant `Microsoft_x64`.

While ECDH exploits much faster execution times the memory consumption of ECDH is higher. Figure 4.6 shows a memory consumption of 7.7 kB (`SIKE_Generic` via `p434`) while ECDH allocates 12.2 kB memory (1.5 times more). Similarly ECDH allocates 1.2 times more memory than SIDH instantiated with `p751` (Figure 4.8).

In order to enable fast and user-friendly cryptography the execution times of cryptographic primitives are essential. ECDH of openssl requires less instructions than all SIDH libraries. While the difference for lower security classes is enormous, the comparison of higher security classes reveal less difference. However, the use of SIDH in a wide range of applications is currently hard to imagine. At the same time research is still ongoing and different optimizations for SIDH were proposed within the last years (see `https://sike.org/`).

4.3.1. Analysis of ECDH execution hotspots

As listed in addenda A.1 the measured execution hotspots for the *openssl* implementation of ECDH differ, since the *openssl* implementation for `secp256r1` is `prime256v1` while `secp384r1` and `secp512r1` are directly implemented in the library [36].

The hotspots of secp256r1 are the low level prime field arithmetic functions __ecp_nistz256_mul_montq (29.9%) and __ecp_nistz256_sqr_montq (18.3%):

1. __ecp_nistz256_mul_montq
   Computation of a montgomery multiplication: $res = a * b * 2^{-256} \ mod \ P$, for integers a, b and P.

2. __ecp_nistz256_sqr_montq
   Computation of a montgomery square: $res = a * a * 2^{-256} \ mod \ P$, for integers a and P.

The measured hotspots for the secp384r1 and secp512r1 are likewise prime field arithmetics: bn_mul_mont claims 67.2% (secp384r1) and 80.0% (secp512r1) of all executed instructions. bn_mod_add_fixed_top demands 6.2% (secp384r1) and 4.3% (secp512r1):

1. bn_mul_mont
   Computation of a montgomery multiplication for *bignum* integers.

2. bn_mod_add_fixed_top
   "BN_mod_add variant that may be used if both a and b are non-negative and less than m."

Similar to the previously described SIDH hotspots the underlying performance bottlenecks of ECDH are related to prime field arithmetic. The well researched ECDH cryptography enables similar algorithms (montgomery multiplication) in order to speed up cryptographic primitives. This results in fast and user-friendly encryption schemes.

4.4. Security Considerations

In order to analyze the given implementations in terms of security, the claim of all libraries to implement security relevant functions in constant time is investigated in subsection 4.4.1. The implemented key sizes of all SIDH libraries and the used ECDH curves will be considered additionally in subsection 4.4.2.

4.4.1. Constant time

Beside the average for $N = 100$ executions ($\bar{x} = \frac{1}{N} \sum_{i=1}^{N} (x_i)$ addenda A.1 also lists the standard deviation of the measured execution time and allocated memory. This standard deviation is computed as $s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2}$. In this section the measured standard deviations are considered to verify if the libraries implement constant time cryptography.
Since the performed public key compression of each *compressed* variant of SIDH depends on the public key itself (rather than on the public key size), they are not implemented in constant time. This is directly visible from addenda A.1.

On the other hand, the standard deviation for the following variants is zero and thus these variants implement constant time cryptography:

- `SIKE_Reference`, `SIKE_Generic` and `SIKE_x64`
- `Microsoft_Generic` and `Microsoft_x64`

The following implementations show deviations in their execution time. Thus, they are not implemented in constant time:

- `ECDH` for the benchmarked curves `secp256r1`, `secp384r1`,`secp512r1` in *openssl*
- `CIRCL_x64`

4.4.2. Key size

This section compares the size of public keys implemented by the SIDH libraries *SIKE*, *PQ-Crypto* and *CIRCL* with modern *openssl* ECDH. The used parameters matching the appropriate NIST security level can be found in section 3.2. Since all SIDH libraries implement the same parameter sets their key sizes are identical. However, *compressed* variants of SIDH benefit from reduced public key sizes, while extending execution time. The key sizes of the used ECDH curves is part of the name, e.g. `secp256r1` exploits 256 bits (256/8 = 32 bytes) as public key. The following table lists the relevant key sizes in bytes:

Algorithm	**SIDH**	**SIDH compressed**	**ECDH**
NIST level 1	330	197	256/8 = 32
NIST level 2	378	225	384/8 = 48
NIST level 3	462	274	-
NIST level 5	564	335	$521/8 \approx 65$

Table 4.1.: Comparison of key sizes in bytes

Shorter public key sizes reduce transmitting and storage costs. The ECDH implementations exploit significantly shorter public keys than SIDH. However, SIDH implements the shortest public key sizes of all quantum-resistant alternatives [31].

5. Conclusion

A. General Addenda

A.1. Detailed Benchmarks

This addenda contains the detailed benchmarks which were measured by the benchmarking suite.
The tables listed contain benchmarking results for the different implementations. For each implementation and each parameter set the listed functions where executed $N = 100$ times. The tables contain the average execution time and average peak memory consumption ($\bar{x} = \frac{1}{N}\sum_{i=1}^{N}(x_i)$). Moreover, the values in brackets are the standard deviation over all measured values. This standard deviation is computed as $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$. All values (except memory) are absolute instruction counts.

A.1.1. Benchmarks for ECDH

Parameter	secp256	secp384	secp521
PrivateKeyA	0 (0)	0 (0)	0 (0)
PublicKeyA	159.418 (0)	10.357.129 (3.556)	24.548.035 (6.482)
PrivateKeyB	0 (0)	0 (0)	0 (0)
PublicKeyB	114.430 (0)	10.307.769 (4.039)	24.499.652 (6.215)
SecretA	652.796 (2)	10.305.471 (3.531)	24.497.519 (6.193)
SecretB	651.270 (2)	10.303.993 (3.882)	24.494.213 (6.054)
Memory in bytes	12.152 (0)	13.984 (0)	16.251 (13)

Table A.1.: Benchmarks for ECDH

Execution hotspots parameter *secp256*:

1. `__ecp_nistz256_mul_montq`: 29.89%
2. `__ecp_nistz256_sqr_montq`: 18.32%
3. `_dl_relocate_object`: 7.27%

Execution hotspots parameter *secp384*:

1. `bn_mul_mont`: 67.23%
2. `bn_mod_add_fixed_top`: 6.28%
3. `bn_mul_mont_fixed_top`: 3.82%

Execution hotspots parameter *secp521*:

1. `bn_mul_mont`: 80.08%
2. `bn_mod_add_fixed_top`: 4.83%
3. `bn_mul_mont_fixed_top`: 2.2%

A.1.2. Benchmarks for Sike Reference

Parameter	434	503	610	751
PrivateKeyA	28.919 (0)	29.020 (0)	34.541 (0)	34.770 (0)
PublicKeyA	1.739.736.057 (0)	2.512.464.770 (0)	4.308.288.592 (0)	7.461.795.045 (0)
PrivateKeyB	29.418 (0)	29.519 (0)	34.617 (0)	35.289 (0)
PublicKeyB	2.352.757.723 (0)	3.435.254.160 (0)	5.790.924.796 (0)	10.556.125.964 (0)
SecretA	18.726.146 (0)	22.075.791 (0)	27.927.622 (0)	35.617.111 (0)
SecretB	43.530.260 (0)	56.573.382 (0)	79.580.027 (0)	118.687.930 (0)
Memory in bytes	11.208 (0)	11.928 (0)	12.904 (0)	13.736 (0)

Table A.2.: Benchmarks for Sike Reference

Execution hotspots parameter *434*:

1. `__gmpn_sbpi1_div_qr`: 10.7%
2. `__gmpn_tdiv_qr`: 9.59%
3. `__gmpn_hgcd2`: 9.29%

Execution hotspots parameter *503*:

1. `__gmpn_sbpi1_div_qr`: 11.08%
2. `__gmpn_hgcd2`: 9.8%
3. `__gmpn_submul_1`: 9.61%

Execution hotspots parameter *610*:

1. `__gmpn_submul_1`: 12.08%
2. `__gmpn_sbpi1_div_qr`: 11.67%
3. `__gmpn_mul_basecase`: 10.13%

Execution hotspots parameter *751*:

1. `__gmpn_submul_1`: 15.21%
2. `__gmpn_mul_basecase`: 11.82%
3. `__gmpn_sbpi1_div_qr`: 11.69%

A.1.3. Benchmarks for Sike Generic

Parameter	434	503	610	751
PrivateKeyA	90 (0)	95 (0)	96 (0)	97 (0)
PublicKeyA	194.932.002 (0)	299.462.858 (0)	618.958.459 (0)	989.778.051 (0)
PrivateKeyB	57 (0)	57 (0)	53 (0)	59 (0)
PublicKeyB	215.702.626 (0)	329.835.556 (0)	616.667.764 (0)	1.111.885.426 (0)
SecretA	158.061.535 (0)	243.950.880 (0)	515.975.033 (0)	813.862.458 (0)
SecretB	181.708.340 (0)	278.459.825 (0)	522.486.909 (0)	947.216.296 (0)
Memory in bytes	7.720 (0)	7.784 (0)	11.288 (0)	13.016 (0)

Table A.3.: Benchmarks for Sike Generic

Execution hotspots parameter *434*:

1. `mp_mul`: 59.45%
2. `rdc_mont`: 31.67%
3. `fp2mul434_mont`: 4.58%

Execution hotspots parameter *503*:

1. `mp_mul`: 59.06%
2. `rdc_mont`: 33.02%
3. `fp2mul503_mont`: 4.07%

Execution hotspots parameter *610*:

1. `mp_mul`: 60.05%
2. `rdc_mont`: 32.8%
3. `fp2mul610_mont`: 4.0%

Execution hotspots parameter *751*:

1. `mp_mul`: 61.06%
2. `rdc_mont`: 32.76%
3. `fp2mul751_mont`: 3.42%

A.1.4. Benchmarks for Sike Generic Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	107 (0)
PublicKeyA	530.693.055 (36.636.736)	800.568.613 (51.179.821)	1.546.560.342 (104.069.626)	2.619.265.805 (178.208.196)
PrivateKeyB	185 (0)	145 (0)	215 (0)	200 (0)
PublicKeyB	430.783.226 (2.956.687)	648.550.068 (5.818.409)	1.209.676.275 (9.989.150)	2.163.046.938 (23.915.892)
SecretA	179.835.016 (2.751)	276.744.609 (2.820)	575.396.837 (4.432)	917.385.785 (6.137)
SecretB	223.183.731 (2.871.219)	342.669.712 (3.247.052)	640.870.333 (5.431.240)	1.141.966.654 (14.089.895)
Memory in bytes	16.968 (0)	19.080 (0)	23.976 (0)	28.008 (0)

Table A.4.: Benchmarks for Sike Generic Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 58.77%
2. `rdc_mont`: 32.15%
3. `fp2mul434_mont`: 4.13%

Execution hotspots parameter *503*:

1. `mp_mul`: 58.4%
2. `rdc_mont`: 33.46%
3. `fp2mul503_mont`: 3.71%

Execution hotspots parameter *610*:

1. `mp_mul`: 59.42%
2. `rdc_mont`: 33.34%
3. `fp2mul610_mont`: 3.61%

Execution hotspots parameter *751*:

1. `mp_mul`: 60.46%
2. `rdc_mont`: 33.29%
3. `fp2mul751_mont`: 3.1%

A.1.5. Benchmarks for Sike x64

Parameter	434	503	610	751
PrivateKeyA	90 (0)	95 (0)	96 (0)	97 (0)
PublicKeyA	18.197.636 (0)	25.309.825 (0)	46.870.491 (0)	67.976.631 (0)
PrivateKeyB	57 (0)	57 (0)	53 (0)	59 (0)
PublicKeyB	20.227.975 (0)	28.024.313 (0)	46.946.162 (0)	76.798.386 (0)
SecretA	14.735.273 (0)	20.595.673 (0)	39.015.534 (0)	55.840.033 (0)
SecretB	17.044.799 (0)	23.672.739 (0)	39.800.369 (0)	65.465.094 (0)
Memory in bytes	8.168 (0)	8.520 (0)	11.392 (0)	13.328 (0)

Table A.5.: Benchmarks for Sike x64

Execution hotspots parameter *434*:

1. `mp_mul`: 52.23%
2. `rdc_mont`: 23.46%
3. `fpsub434`: 4.22%

Execution hotspots parameter *503*:

1. `mp_mul`: 49.88%
2. `rdc_mont`: 27.18%
3. `fpsub503`: 4.12%

Execution hotspots parameter *610*:

1. `mp_mul`: 51.51%
2. `rdc_mont`: 28.57%
3. `fpsub610`: 3.72%

Execution hotspots parameter *751*:

1. `mp_mul`: 53.34%
2. `rdc_mont`: 27.94%
3. `fpsub751`: 3.81%

A.1.6. Benchmarks for Sike x64 Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	107 (0)
PublicKeyA	51.478.796 (3.842.012)	70.633.473 (4.623.263)	120.698.795 (6.557.301)	184.983.750 (10.400.022)
PrivateKeyB	142 (0)	144 (0)	179 (0)	188 (0)
PublicKeyB	41.648.989 (345.077)	56.629.973 (517.293)	94.418.639 (796.824)	152.512.949 (1.142.155)
SecretA	17.062.490 (1.558)	23.745.084 (2.055)	44.067.290 (2.871)	63.697.941 (4.288)
SecretB	21.428.968 (241.953)	29.717.372 (262.099)	49.633.365 (394.707)	80.095.836 (886.898)
Memory in bytes	19.240 (0)	21.608 (0)	27.264 (0)	31.936 (0)

Table A.6.: Benchmarks for Sike x64 Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 50.21%
2. `rdc_mont`: 23.14%
3. `fpsub434`: 4.22%

Execution hotspots parameter *503*:

1. `mp_mul`: 47.88%
2. `rdc_mont`: 26.74%
3. `fpsub503`: 4.1%

Execution hotspots parameter *610*:

1. `mp_mul`: 49.73%
2. `rdc_mont`: 28.31%
3. `fpsub610`: 3.75%

Execution hotspots parameter *751*:

1. `mp_mul`: 51.71%
2. `rdc_mont`: 27.79%
3. `fpsub751`: 3.82%

A.1.7. Benchmarks for CIRCL x64

Parameter	434	503	751
PrivateKeyA	671 (0)	671 (0)	671 (0)
PublicKeyA	27.173.063 (12.785)	30.588.687 (14.339)	92.516.170 (11.132)
PrivateKeyB	672 (0)	672 (0)	672 (0)
PublicKeyB	28.959.178 (431)	32.771.303 (314)	103.101.351 (244)
SecretA	21.089.046 (2.101)	23.926.495 (1.461)	75.130.980 (636)
SecretB	24.364.188 (324)	27.627.692 (360)	87.853.968 (337)
Memory in bytes	21.654 (1.022)	21.691 (896)	21.663 (1.055)

Table A.7.: Benchmarks for CIRCL x64

Execution hotspots parameter *434*:

1. `p434.mulP434:` 47.28%
2. `p434.rdcP434:` 22.93%
3. `p434.subP434:` 5.81%

Execution hotspots parameter *503*:

1. `p503.mulP503:` 41.39%
2. `p503.rdcP503:` 23.04%
3. `p503.subP503:` 8.39%

Execution hotspots parameter *751*:

1. `p751.mulP751:` 56.16%
2. `p751.rdcP751:` 21.72%
3. `p751.subP751:` 6.02%

A.1.8. Benchmarks for Microsoft Generic

Parameter	434	503	610	751
PrivateKeyA	86 (0)	91 (0)	91 (0)	91 (0)
PublicKeyA	195.425.484 (0)	300.437.100 (0)	620.117.514 (0)	992.618.213 (0)
PrivateKeyB	53 (0)	53 (0)	48 (0)	53 (0)
PublicKeyB	216.131.501 (0)	330.771.955 (0)	617.536.325 (0)	1.114.734.257 (0)
SecretA	158.459.759 (0)	244.758.599 (0)	516.977.970 (0)	816.142.131 (0)
SecretB	182.033.988 (0)	279.212.025 (0)	523.130.069 (0)	949.500.928 (0)
Memory in bytes	8.040 (0)	8.360 (0)	11.784 (0)	13.624 (0)

Table A.8.: Benchmarks for Microsoft Generic

Execution hotspots parameter *434*:

1. `mp_mul`: 60.55%
2. `rdc_mont`: 31.9%
3. `fp2mul434_mont`: 4.56%

Execution hotspots parameter *503*:

1. `mp_mul`: 60.12%
2. `rdc_mont`: 33.25%
3. `fp2mul503_mont`: 3.96%

Execution hotspots parameter *610*:

1. `mp_mul`: 61.24%
2. `rdc_mont`: 32.54%
3. `fp2mul610_mont`: 4.02%

Execution hotspots parameter *751*:

1. `mp_mul`: 62.22%
2. `rdc_mont`: 32.44%
3. `fp2mul751_mont`: 3.43%

A.1.9. Benchmarks for Microsoft Generic Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	105 (0)
PublicKeyA	354.807.131 (28.640.848)	548.929.246 (48.233.475)	1.058.952.592 (69.882.412)	1.767.280.284 (114.624.073)
PrivateKeyB	239 (0)	233 (0)	206 (0)	314 (0)
PublicKeyB	340.645.715 (4.510.751)	513.777.907 (5.790.443)	956.709.613 (11.260.973)	1.731.204.963 (18.353.464)
SecretA	177.456.197 (1.970)	274.002.754 (1.875)	569.195.853 (3.954)	908.459.651 (5.191)
SecretB	201.752.486 (871)	309.529.792 (1.000)	577.403.156 (1.459)	1.043.916.680 (1.618)
Memory in bytes	69.576 (0)	89.896 (0)	134.600 (0)	193.336 (0)

Table A.9.: Benchmarks for Microsoft Generic Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 59.82%
2. `rdc_mont`: 32.48%
3. `fp2mul434_mont`: 4.06%

Execution hotspots parameter *503*:

1. `mp_mul`: 59.26%
2. `rdc_mont`: 33.89%
3. `fp2mul503_mont`: 2.89%

Execution hotspots parameter *610*:

1. `mp_mul`: 60.56%
2. `rdc_mont`: 33.26%
3. `fp2mul610_mont`: 3.55%

Execution hotspots parameter *751*:

1. `mp_mul`: 61.48%
2. `rdc_mont`: 33.03%
3. `fp2mul751_mont`: 2.33%

A.1.10. Benchmarks for Microsoft x64

Parameter	434	503	610	751
PrivateKeyA	86 (0)	91 (0)	91 (0)	91 (0)
PublicKeyA	16.733.523 (0)	23.373.795 (0)	44.826.467 (0)	64.976.610 (0)
PrivateKeyB	53 (0)	53 (0)	48 (0)	53 (0)
PublicKeyB	18.587.638 (0)	25.858.758 (0)	44.876.850 (0)	73.377.700 (0)
SecretA	13.529.422 (0)	18.993.109 (0)	37.346.394 (0)	53.326.381 (0)
SecretB	15.655.268 (0)	21.831.732 (0)	38.025.823 (0)	62.514.039 (0)
Memory in bytes	8.936 (0)	9.048 (0)	12.944 (0)	15.008 (0)

Table A.10.: Benchmarks for Microsoft x64

Execution hotspots parameter *434*:

1. `mp_mul`: 55.81%
2. `rdc_mont`: 23.82%
3. `0x000000000000cd3c`: 4.56%

Execution hotspots parameter *503*:

1. `mp_mul`: 52.33%
2. `rdc_mont`: 28.27%
3. `0x000000000000d8e4`: 4.38%

Execution hotspots parameter *610*:

1. `mp_mul`: 53.86%
2. `rdc_mont`: 29.88%
3. `0x000000000000f256`: 3.86%

Execution hotspots parameter *751*:

1. `mp_mul`: 55.83%
2. `rdc_mont`: 29.24%
3. `0x000000000000fefd`: 3.62%

A.1.11. Benchmarks for Microsoft x64 Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	105 (0)
PublicKeyA	32.099.997 (2.309.282)	44.418.194 (3.712.967)	80.996.244 (6.287.779)	123.157.832 (12.599.138)
PrivateKeyB	149 (0)	152 (0)	187 (0)	200 (0)
PublicKeyB	30.784.457 (316.772)	41.848.170 (319.398)	72.151.350 (669.781)	118.144.325 (1.757.118)
SecretA	15.538.409 (588)	21.689.738 (483)	42.096.902 (1.350)	60.202.926 (1.955)
SecretB	17.949.468 (571)	24.856.517 (656)	42.885.495 (895)	69.857.944 (1.243)
Memory in bytes	69.960 (0)	90.640 (0)	135.216 (0)	193.872 (0)

Table A.11.: Benchmarks for Microsoft x64 Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 52.82%
2. `rdc_mont`: 23.28%
3. `0x00000000000247dc`: 3.56%

Execution hotspots parameter *503*:

1. `mp_mul`: 49.75%
2. `rdc_mont`: 27.75%
3. `0x0000000000026694`: 3.45%

Execution hotspots parameter *610*:

1. `mp_mul`: 51.53%
2. `rdc_mont`: 29.48%
3. `0x000000000002e046`: 3.09%

Execution hotspots parameter *751*:

1. `mp_mul`: 53.4%
2. `rdc_mont`: 28.99%
3. `0x0000000000032f6d`: 2.8%

List of Figures

List of Tables

Bibliography

[1] C. Eckert. *IT-Sicherheit*. Berlin, Boston: De Gruyter Oldenbourg, 21 Aug. 2018. ISBN: 978-3-11-056390-0. DOI: https://doi.org/10.1515/9783110563900. URL: https://www.degruyter.com/view/title/530046.

[2] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[3] D. Wätjen. *Kryptographie*. Springer, 2018.

[4] J. Randall, B. Kaliski, J. Brainard, and S. Turner. "Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)". In: *Proposed Standard* 5990 (2010).

[5] M. A. Nielsen and I. Chuang. *Quantum computation and quantum information*. 2002.

[6] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. *Report on post-quantum cryptography*. Vol. 12. US Department of Commerce, National Institute of Standards and Technology, 2016.

[7] D. A. Lidar and T. A. Brun. *Quantum error correction*. Cambridge university press, 2013.

[8] F. Boudot, P. Gaudry, A. Guillevic, N. Heninger, E. Thome, and P. Zimmermann. *795-bit factoring and discrete logarithms*.

[9] A. Beutelspacher, H. B. Neumann, and T. Schwarzpaul. "Der diskrete Logarithmus, Diffie-Hellman-Schlüsselvereinbarung, ElGamal-Systeme". In: *Kryptografie in Theorie und Praxis*. Springer, 2010, pp. 132–144.

[10] P. W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

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

[12] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang. "The impact of quantum computing on present cryptography". In: *arXiv preprint arXiv:1804.00200* (2018).

[13] D. P. Chi, J. W. Choi, J. San Kim, and T. Kim. "Lattice based cryptography for beginners." In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 938.

[14] J. Hoffstein, J. Pipher, and J. H. Silverman. "NTRU: A ring-based public key cryptosystem". In: *International Algorithmic Number Theory Symposium*. Springer. 1998, pp. 267–288.

[15] C. Gentry and D. Boneh. *A fully homomorphic encryption scheme*. Vol. 20. 9. Stanford university Stanford, 2009.

[16] J. Hartmanis. "Computers and intractability: a guide to the theory of NP-completeness (michael r. garey and david s. johnson)". In: *Siam Review* 24.1 (1982), p. 90.

[17] J. Ding and A. Petzoldt. "Current state of multivariate cryptography". In: *IEEE Security & Privacy* 15.4 (2017), pp. 28–36.

[18] J. Ding and D. Schmidt. "Rainbow, a new multivariable polynomial signature scheme". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2005, pp. 164–175.

[19] D. J. Bernstein and T. Lange. "Post-quantum cryptography". In: *Nature* 549.7671 (2017), pp. 188–194.

[20] R. J. McEliece. "A public-key cryptosystem based on algebraic". In: *Coding Thv* 4244 (1978), pp. 114–116.

[21] G. Becker. "Merkle signature schemes, merkle trees and their cryptanalysis". In: *Ruhr-University Bochum, Tech. Rep* (2008).

[22] R. C. Merkle. *Secrecy, authentication, and public key systems.* Stanford University, 1979.

[23] D. Jao and L. De Feo. "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies". In: *International Workshop on Post-Quantum Cryptography*. Springer. 2011, pp. 19–34.

[24] L. De Feo. "Supersingular Isogeny Key Encapsulation". In: *NIST round 3 submission*. 2020.

[25] D. Urbanik. "A Friendly Introduction to Supersingular Isogeny Diffie-Hellman". In: (2017).

[26] C. Costello. "Supersingular Isogeny Key Exchange for Beginners". In: *International Conference on Selected Areas in Cryptography*. Springer. 2019, pp. 21–50.

[27] C. Costello and C. AIMSCS. "A gentle introduction to isogeny-based cryptography". In: *Tutorial Talk at SPACE* (2016).

[28] S. Jaques and J. M. Schanck. "Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE". In: *Annual International Cryptology Conference*. Springer. 2019, pp. 32–61.

[29] S. Tani. "Claw finding algorithms using quantum walk". In: *Theoretical Computer Science* 410.50 (2009), pp. 5285–5297.

[30] P. C. Van Oorschot and M. J. Wiener. "Parallel collision search with cryptanalytic applications". In: *Journal of cryptology* 12.1 (1999), pp. 1–28.

[31] B. Koziel, R. Azarderakhsh, and M. M. Kermani. "A high-performance and scalable hardware architecture for isogeny-based cryptography". In: *IEEE Transactions on Computers* 67.11 (2018), pp. 1594–1609.

[32] Microsoft. *PQCrypto-SIDH*. `https://github.com/microsoft/PQCrypto-SIDH`. 2020.

[33] Cloudflare. *CIRCL*. https://github.com/cloudflare/circl. 2020.

[34] K. Kwiatkowski and A. Faz-Hernández. *Introducing CIRCL: An Advanced Cryptographic Library*. July 2019. URL: https://blog.cloudflare.com/introducing-circl/.

[35] *GoLang Wiki Compiler Optimizations*. https://github.com/golang/go/wiki/CompilerOptimizations. Accessed: 2020-09-25.

[36] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk. "Elliptic curve cryptography subject public key information". In: *RFC 5480 (Proposed Standard)* (2009).

[37] D. R. Brown. "Sec 2: Recommended elliptic curve domain parameters". In: *Standars for Efficient Cryptography* (2010).