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:	15.02.2021

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

Munich, 15.02.2021 Jonas Hagg

Acknowledgments

Abstract

This thesis describes currently available Supersingular Isogeny Diffie-Hellman (SIDH) impelementations: *SIKE*, *CIRCL* and *SIKE for Java*. The performance of *SIKE* and *CIRCL* are benchmarked in detail: While the developed benchmarking suite indicates *SIKE* as the library executing the least amount of instructions for *x64 optimized* algorithms, *CIRCL* executes less instructions for *generic optimized* algorithms. For all implementations *SIKE* allocates less memory. The comparison with a modern Elliptic Curve Diffie-Hellman library (*OpenSSL*) demonstrates the limitations of current SIDH algorithms in terms of execution time for a single key exchange.

Contents

1. Introduction

This thesis benchmarks several Supersingular Isogeny Diffie-Hellman (SIDH) implementations. Cryptography based on supersingular isogenies is a candidate to replace state-of-the-art asymmetric encryption primitives in the upcoming age of quantum-computers. Especially SIDH might replace modern Diffie-Hellman key exchange algorithms in future.
The goal of this thesis is the comparison between existing SIDH implementations. For a complete analysis of the current state of SIDH, this upcoming technology is also compared to widely used Elliptic Curve Diffie-Hellman (ECDH) protocols.
In order to generate comparable benchmarks, a benchmarking suite is developed for this thesis. The SIDH implementations *SIKE* and *CIRCL* are integrated into this benchmarking suite to obtain reliable benchmarking results.

After this introduction, Chapter 2 firstly introduces the necessary technical background. Besides the introduction of basic encryption and key exchange protocols, the chapter illustrates the influence of upcoming quantum computing to modern cryptography. Finally, the term Supersingular Isogeny Diffie-Hellman (SIDH) is introduced in detail.
Chapter 3 presents currently available libraries implementing SIDH: *SIKE*, *CIRCL* and *Sike for Java*. Their implemented primitives, optimized versions, parameter sets and APIs are summarized and compared. Moreover the relation between the similar libraries SIKE and PQCrypto-SIDH is investigated.
In Chapter 4 the benchmarking suite – developed in the scope of this thesis – is introduced. Besides the usage and the internal operations of the software, the chapter also reveals and justifies the applied benchmarking methodologies.
Chapter 5 provides the results measured by the benchmarking suite. The SIDH libraries *SIKE* and *CIRCL* are are compared based on their available implementations. Furthermore, the chapter takes ECDH in consideration to compare SIDH with currently deployed technologies.
The final Chapter 6 summarizes the major results and describes limitations of this thesis.

2. Background

This opening chapter covers the technical background needed to read and understand this thesis. Besides a short introduction into modern cryptography schemes and advanced key exchange concepts, the emergence and consequences of quantum computers are explained. Finally, this chapter discusses isogeny-based cryptography – the theoretical background of this thesis.

In modern cryptography one can distinguish 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 owner has knowledge about the private key.

Symmetric Cryptography

Figure 2.1 shows a simple symmetric encryption scheme. First, a plaintext is encrypted using a symmetric encryption algorithm and a secret key. The resulting ciphertext 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 that insecure channel is critical: Somehow the symmetric key needs to be transported securely to the receiver of the ciphertext.

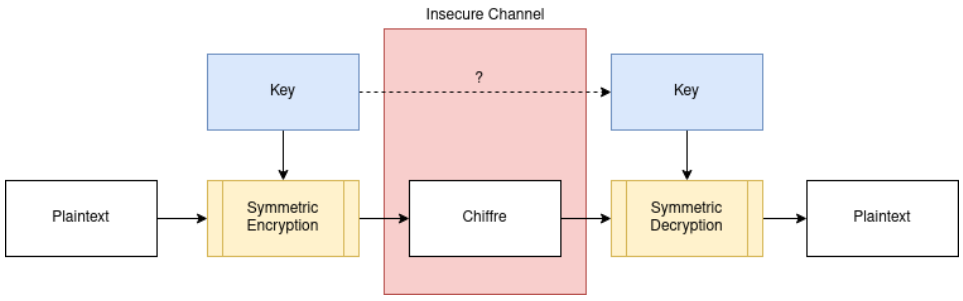

Figure 2.1.: Simple symmetric encryption scheme: Encryption and decryption algorithm use the same key.

In the following, 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 owner and is kept secret. Figure 2.2 shows an example for asymmetric encryption. Assume that Alice wants to send encrypted data to Bob. Therefore, Bob created a key pair and published his public key. Alice requests Bob's 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 thesis, the term *public-key encryption*, *PKE* and *public-key algorithm* are used as a synonyms for asymmetric cryptography.

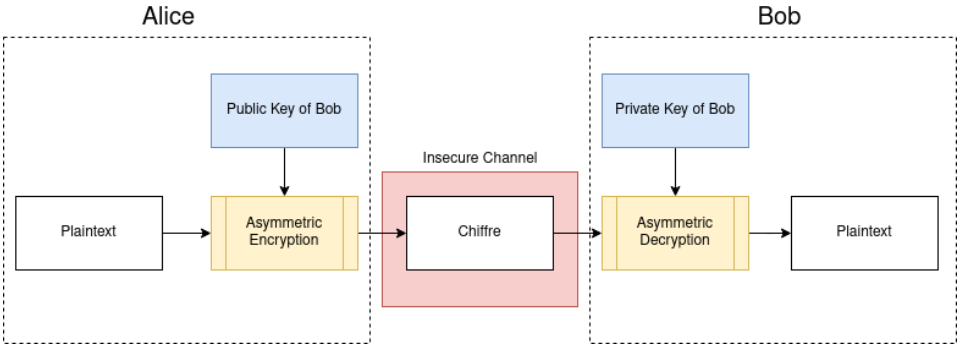

Figure 2.2.: Asymmetric encryption scheme: Encryption and decryption algorithm use different keys.

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 *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, two major key exchange protocols are available.

2.1. Key Exchange

This section describes and compares the major protocols to establish a shared secret between two communicating subjects: The *Diffie-Hellman Key-Exchange* and a *Key Encapsulation Mechanism*.

2.1.1. Diffie-Hellman Key-Exchange

The Diffie-Hellman key exchange was introduced by Whitfield Diffie and Martin Hellman in 1976 [2]. This protocol creates a shared secret between two subjects. The resulting shared key of the protocol is calculated decentralized and is never transported through 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 $A = g^a \bmod p$.
3. Bob choses a random $b \in \{1, 2, ..., p-2\}$ as private key.
4. Bob calculates the public key $B = g^b \bmod p$.
5. Alice and Bob exchange their public keys A and B.
6. Alice calculates:

$$\begin{aligned} k_{AB} &= B^a \bmod p \\ &= (g^b mod p)^a \bmod p \\ &= g^{ba} \bmod p \\ &= g^{ab} \bmod p \end{aligned} \tag{2.1}$$

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

7. Bob calculates:

$$\begin{aligned} k_{AB} &= A^b \bmod p \\ &= (g^a \bmod p)^b \bmod p \\ &= g^{ab} \bmod p \end{aligned} \tag{2.2}$$

8. Alice and Bob created the shared secret k_{AB}. Note that only the public keys of Alice and Bob were sent through an insecure channel. The generated secret was calculated locally by Alice and Bob, respectively.

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

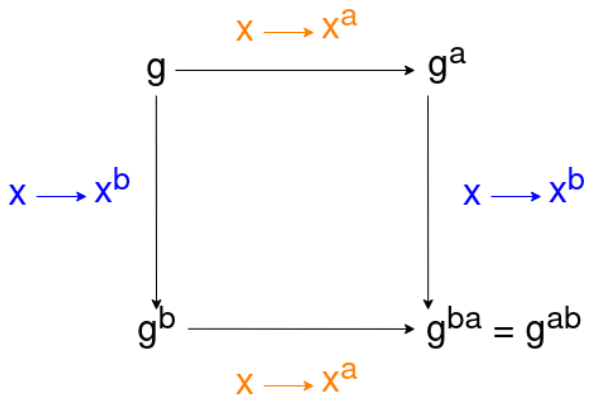

Figure 2.3.: Diffie-Hellman diagram - both paths lead to the same result.

Security

The security of the Diffie-Hellman protocol is based on the challenge: Given the public generator element g, the public key of Alice g^a and the public key of Bob g^b, compute g^{ab}. If an attacker wants to compute g^{ab}, she needs to compute the private keys a and b of Alice and Bob. Since only the public keys $A = g^a$ and $B = g^b$ are exchanged, it suffices to compute:

$$\begin{aligned} b &= \log_g B \bmod p \\ a &= \log_g A \bmod p \end{aligned}$$

Hence, the security of the classical Diffie-Hellman key exchange is based on the discrete logarithm problem considered difficult to be solved by classical computers (see section 2.2). However, if an attacker is able to solve this challenge, this would neither compromise any keys from the past nor any future keys of the communication (this is called *perfect forward*

secrecy, short *PFS* [1]). Another Diffie-Hellman handshake would challenge the attacker with the discrete logarithm problem again. Thus, the Diffie-Hellman key exchange may be used as efficient PFS protocol when keys are renewed regularly.
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 the classical Diffie-Hellman protocol [1].
Note that the introduced protocol does not authenticate 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].

2.1.2. 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 illustrated by RSA-KEM using RSA key pairs to transmit a shared secret of length n 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}:

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

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:

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

Security

If an attacker wants to compute k_{AB} it is necessary to break the RSA encryption $RSA_{enc}(m, e_{Bob})$ in order to reveal m. Applying the inverse mapping function h^{-1} then yields the secret key k_{AB}. In order to break the RSA encryption it is sufficient to compute the private key of Bob given his public key. This computation is assumed to be equally demanding as solving the factorization problem (see section 2.2) for big numbers [5].
Note that once an attacker was able to compromise Bobs private key, all following exchanged shared secrets k_{AB} are compromised as well. Thus, this protocol does not ensure perfect forward secrecy (PFS).

2.1.3. Differences

Both presented protocols securely share a symmetric encryption key between two communicating subjects. However, there are some differences between KEM and Diffie-Hellman. Firstly, while KEMs transmit a shared secret from one subject to another, the calculation in the Diffie-Hellman protocol is decentralized. Thus, the shared secret will never be send through an insecure channel.
Secondly, KEM relies on a long-term asymmetric key pair which is used to encapsulate and decapsulate 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. Thus, DH key exchanges might be used as the basis of a PFS protocol.
Thirdly, KEMs can be used offline: Since one subject (e.g the sending subject) chooses the secret key, data calculation and encryption can be performed without any previous communication with the receiver. Thus, no synchronization between the subjects is necessary to exchange encrypted data.

2.2. 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 Turing machine. In contrast to *classical computer* the term *quantum computer* describes a machine using quantum mechanical phenomena to perform computations. It is important to note that quantum computers can simulate classical computers [6]. In addition, classical computer are able to simulate quantum computers with exponential time overhead [6]. Thus, classical and quantum computers can calculate the same class of functions. However, quantum computers enable operations allowing much faster computation in some cases [6].
In the past, scientists queried whether large-scale quantum computer are physically possible. It was argued that the underlying quantum states are too fragile and hard to control [7]. Today, quantum error correction codes are known, putting large-scale quantum computers within the realms of possibility [8]. However, it is still a major engineering challenge from a laboratory approach to a general-purpose quantum computer that involves thousands or millions of logical qubits [7].

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) [7]. While these problems are theoretically solvable, the computation on classical computers claim 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 corresponds to running a CPU for a full year) [9]. In the following, the discrete logarithm problem and the factorization problem are described.

Discrete Logarithm Problem

The discrete logarithm problem consists in the following challenge [10]: Given a prim p and two integers g and y. Find an integer x, such that

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

Up to this date, it remains still unknown if a classical computer is able to compute the general discrete logarithm problem in polynomial time. Thus, the discrete logarithm problem is considered difficult to be solved by classical computers [10]. This assumption makes the discrete logarithm problem an attractive basis for various cryptographic primitives: DSA, ElGamal, classical Diffie-Hellman, and Elliptic Curve Diffie-Hellman employ 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 their respective product:

$$n = p \cdot q$$

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.

2.2.1. Impact of Quantum Computers on Cryptography

As stated above, quantum computers enable new operations which accelerate certain algorithms. Two quantum algorithms which have enormous consequences on modern cryptography are *Shor's algorithm* [11] and *Grover's algorithm* [12].

Shor's Algorithm

Peter Shor published *"Algorithms for quantum computation: discrete logarithms and factoring"* in 1994 [11]. In this publication he demonstrated that the factorization problem and the discrete logarithm problem can be solved in polynomial time on quantum computers. Both problems form the basis of many public-key systems (RSA, DH, ECDH, ...) used intensively in modern communication systems. Hence, a quantum computer running *Shor's algorithm* would qualify for the assumption of most asymmetric encryption schemes and thus break their security.

Grover's Algorithm

The second algorithm impacting computer security was published by Lov Grover in 1996 (*"A fast quantum mechanical algorithm for database search"*, [12]) - also referred to as *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 $\mathcal{O}(\sqrt{N})$ [6].
In contrast to public-key systems that rely on hard mathematical problems, symmetric encryption schemes rely in particular on the secrecy of a randomly generated key. Thus, to break symmetric encryption, one can perform a brute-force attack on the symmetric key. *Grover's algorithm* offers a square root speed-up on classical brute-force attacks [13]. Assume a randomly generated n-bit key. A classical brute force algorithm takes $\mathcal{O}(2^n)$ steps, which is considered to be safe for a big n (e.g. $n = 128$). *Grover's algorithm* speeds up this attack to $\mathcal{O}(\sqrt{2^n}) = \mathcal{O}(2^{n/2})$ steps [13]. For $n = 128$ this results in a security level of 64-bit ($2^{128/2} = 2^{64}$) [13]. Note that NIST considers a security-level of at least 112-bits as secure [14]. However, the complexity is still exponential and with a growing key size n the security can be increased further (e.g. doubling key size n is sufficient to restore previous security). Thus, *Grover's*

algorithm forces symmetric encryption schemes to increase their key size in order to stay secure.

Summing up, quantum computers make use of quantum mechanical phenomena in order to solve mathematical problems which are assumed to be difficult for classical 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"* [7] demonstrates 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	Public key	Signatures, key exchange	No longer secure
ECDH, ECDSA	Public key	Signatures, key exchange	No longer secure

Table 2.1.: Impact of quantum computers on modern encryption schemes (adopted from [7]).

In contrast to this development in modern cryptography NIST states [7]:

> *"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 [15]. 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 this standardization process was announced. Different approaches for quantum-resistant algorithms have been proposed. This thesis focus on isogeny-based cryptography (section 2.3). Other classes of post-quantum cryptography are described for completeness in the following section 2.2.2.

2.2.2. Classes of Post-Quantum Cryptography

This section provides an overview over important post-quantum cryptography classes: Lattice-based, multivariate and code-based cryptography as well as hash-based signatures are briefly

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 that are considered difficult to be solved even by quantum computers [16]. In 1998, NTRU was published as the first public-key system based on lattices [17]. 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 [18].
Lattice-based cryptography is characterized by simplicity and efficiency [7]. However, lattices encryption schemes have problems to prove security against known cryptoanalysis [7].

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 $\mathbb{F}$. The proof that solving systems of multivariate polynomials are NP-hard [19] is the basis of their security. 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 [20], e.g. the Rainbow signature scheme [21].

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 [22]. McEliece, published in 1978, was the first of those systems and it has not been broken until today [23]. On the other hand, code-based cryptography requires large key sizes [22].
Besides asymmetric cryptography, code-based schemes have been proposed for digital signatures, random number generators and cryptographic hash functions [22].

Hash-based Signatures

Hash-based signatures refer to 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 [22]. Since hash functions are widely deployed in modern computer systems, the security of hash-based signatures is well understood [7].

[2] A lattice l is a subgroup of $\mathbb{R}^n$. In the context of cryptography usually integer lattices are considered: $l \subseteq \mathbb{Z}^n$ [16].

The initially developed One-Time Signatures have the downside that a new public key pair is needed for each signature [24]. In 1979, Merkle introduced the Merkle Signature Scheme (MSS) which uses one public key for multiple signatures [25]. Further improvements of MSS introduced public keys usable for 2^{80} signatures. However, this also leads to longer signature sizes [24].

2.3. Isogeny-based Cryptography

Isogeny-based cryptography was proposed in 2011 as a new cryptographic system that might resists quantum computing [26]. Besides describing isogeny-based cryptography, the authors also provided a reference implementation for a PKE and a KEM based on isogenies. This implementation is called *SIKE* [27]. Isogeny-based cryptography benefits from small key sizes compared to other post-quantum cryptography classes, however, their performance is comparatively slow [27]. The security of these primitives is based on finding isogenies between supersingular elliptic curves.
In the following, the problem is firstly roughly illustrated. It is not indented to provide exact mathematical details about isogeny-based cryptography, since that would be beyond the scope of this thesis. However, one might become an intuition for supersingular isogenies. Subsequently, the central component of isogeny-based cryptography - namely the Supersingular Isogeny Diffie-Hellman (SIDH) - is described. Finally, details of the reference implementation SIKE are given and the security of SIDH is considered.

2.3.1. Mathematics

This section is adopted from "A Friendly Introduction to Supersingular Isogeny Diffie-Hellman" [28] and "Supersingular Isogeny Key Exchange for Beginners" [29].
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. Essentially, this means to construct a new elliptic curve $E\backslash S$. Besides this new curve, the procedure also yields a function $\phi_S : E \rightarrow E\backslash S$ which is called an *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 follows:

1. Start with a known curve E and build an isogeny to an 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.

Usually, in asymmetric asymmetric cryptography the hard mathematical problem consists in computing the private key while knowing the public key. To be more precise: Find the isogeny $\phi_A : E \to E_A$ while knowing curves E and E_A is considered to be a quantum-resistant challenge. In literature, this is formally defined as *SIDH problem* [27].
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, isogeny-based cryptography creates a shared secret via a Diffie-Hellman like procedure. This is called *Supersingular Isogeny Diffie Hellman (SIDH)*.

2.3.2. Supersingular Isogeny Diffie-Hellman (SIDH)

In the previous section the underlying mathematical idea of isogeny-based cryptography 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. The SIDH key exchange is represented by the following protocol.
Starting point of SIDH is a publicly known supersingular elliptic curve E.

1. Alice creates an isogeny ϕ_A (private key) which leads to curve E_A (part of Alice's public key).
2. Bob creates an isogeny ϕ_B (private key) which leads to curve E_B (part of Bob's 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 share the common secret E_{AB}.

Figure 2.3 above showed the commutative property of the classical Diffie-Hellman protocol. The same diagram can be drawn for the Diffie-Hellman based on supersingular isogenies (see Figure 2.4):

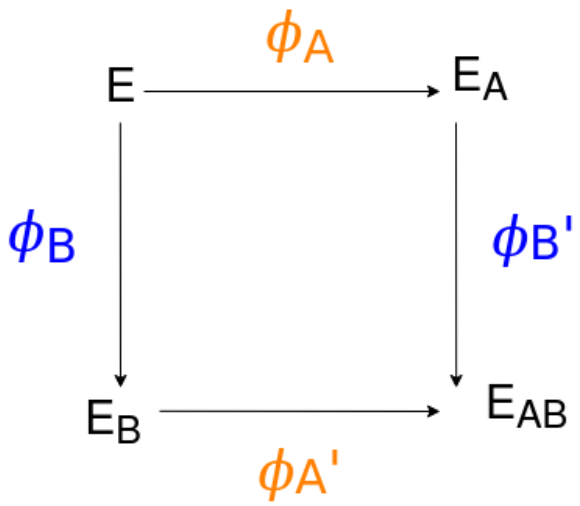

Figure 2.4.: Supersingular Isogeny Diffie-Hellman procedure - both paths lead to the same result E_{AB}. Note the different isogenies ϕ'_A and ϕ'_B that are applied in each second step of the diagram. The reason for this lies in the mathematics of supersingular isogenies. In order to construct ϕ'_X or ϕ'_B the public key of each communication partner provides additional information besides the curve E_A or E_B. [30]

2.3.3. Implementation Details

The implementation details described in this subsection are based on *SIKE* [27], the first proposed supersingular isogeny-based cryptography. The following explanations also illustrate the connection between SIDH and isogeny-based PKE and KEM.
The reference implementation of SIKE provides two fundamental functions: *isogen* and *isoex*. Both are used, to implement the previously introduced SIDH algorithm. Note that the secret key *sk* is represented as a random integer. Moreover, the used parameters that represent the applied supersingular curve are also passed to *isogen* and *isoex*. For simplicity, this is not listed here, however the exact parameter sets for all curves are listed in the specification of SIKE [27].

Function	Input	Output
isogen	parameter set *param* party *p* secret key *sk*	public key pk
isoex	parameter set *param* party *p* secret key *sk* public key *pk*	shared secret *sec*

Table 2.2.: Core functions of the SIKE reference implementation.

The function *isogen* takes a secret key (random integer), the parameter set and the party (Alice or Bob) as input and generates the public key. The shared secret is generated by *isoex* taking the own secret key and the foreign public key as input. Additionally *isoex* also expect the used parameter set and the party of the key exchange. The party needs to be specified since Alice and Bob do not use the same logic to generate their keys and the shared secret. The reason for this lies in the mathematical foundation of isogeny based cryptography. The SIDH key exchange procedure with respect to *isogen* and *isoex* works as follows:

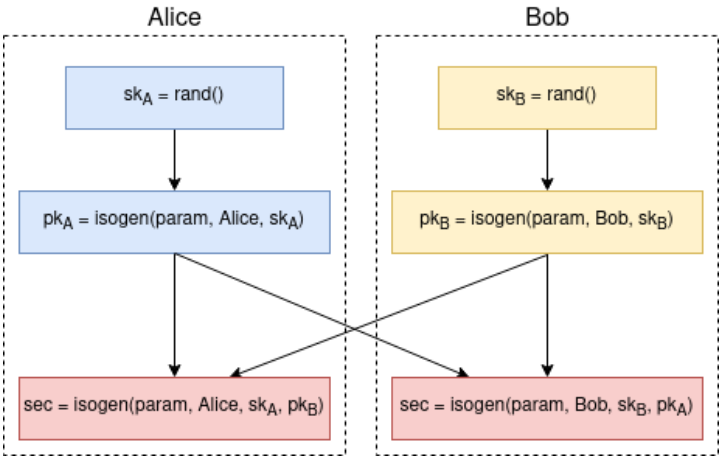

Figure 2.5.: SIDH based on *isogen* and *isoex*: After generating a random secret key *sk* each subject computes its public key *pk*. After the exchange of public keys each subject finally calculates the shared secret *sec*. It is important that both parties use the same parameter set *param*

Besides this key exchange algorithm, SIKE provides a complete asymmetric encryption scheme and a key encapsulation mechanism [27]. 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 2.6) consists of three algorithms. The authors of SIKE state, that their PKE scheme is a modification of the classical hashed ElGamal scheme by replacing classical Diffie-Hellman procedure with quantum-secure SIDH [27].

1. *Gen* generates a key pair (sk_A,pk_A). Firstly, the private key is chosen at random. The public key is then derived from the random private key using *isogen*.

2. *Enc* encrypts a given plaintext m using a foreign public key pk_B and a random r. Internally, the random r is treated as private key: c_0 is derived from r via *isogen*. A secret is then generated by *isoex* using the foreign public key pk_B and the private key r. The hash of this secret $F(sec)$ is finally XORed with the given plaintext m and saved in c_1. The function returns c_0 and c_1.

3. *Dec* decodes a given ciphertext (c_0,c_1) using the secret key sk_B. Firstly, a secret is derived by *isoex* using the private key sk_B and foreign public key c_0. The hash of this secret $F(sec)$ is XORed with the ciphertext c_1 returning the plaintext m. The function returns m.

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

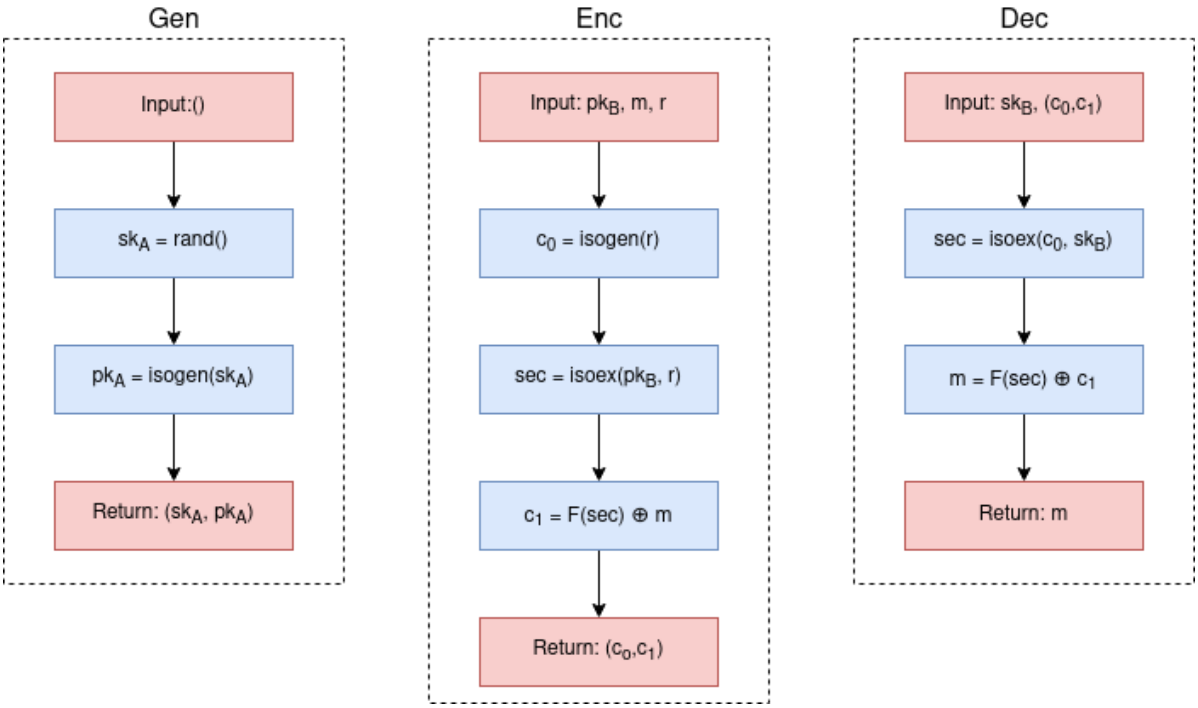

Figure 2.6.: Isogeny-based public-key encryption (PKE) scheme.

Isogeny-based KEM

The construction of the isogeny-based key encapsulation mechanism (KEM, Figure 2.7) consists of three algorithms. The authors of SIKE state to build a *IND-CCA* KEM from their PKE [27] (by applying a slight modification of the transformation published by Hofheinz, Hüvelsmann and Klitz [31]). This essentially means, that an attacker can not distinguish two ciphertexts created by the KEM encryption function *Encaps*. The following algorithms describe how the isogeny-based KEM is created from the isogeny-based PKE described above. The goal of this construction is to build an *IND-CCA* KEM (see [31] for details).

1. *KeyGen* generates a key pair (sk_A,pk_A) and a secret s. The private key sk and the secret s are chosen at random. The public key is derived from the random private key using *isogen*.

2. *Encaps* takes a given public key pk_B 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. *Decaps* takes a ciphertext c and the output of *Gen* as input in order to retrieve the shared secret K.

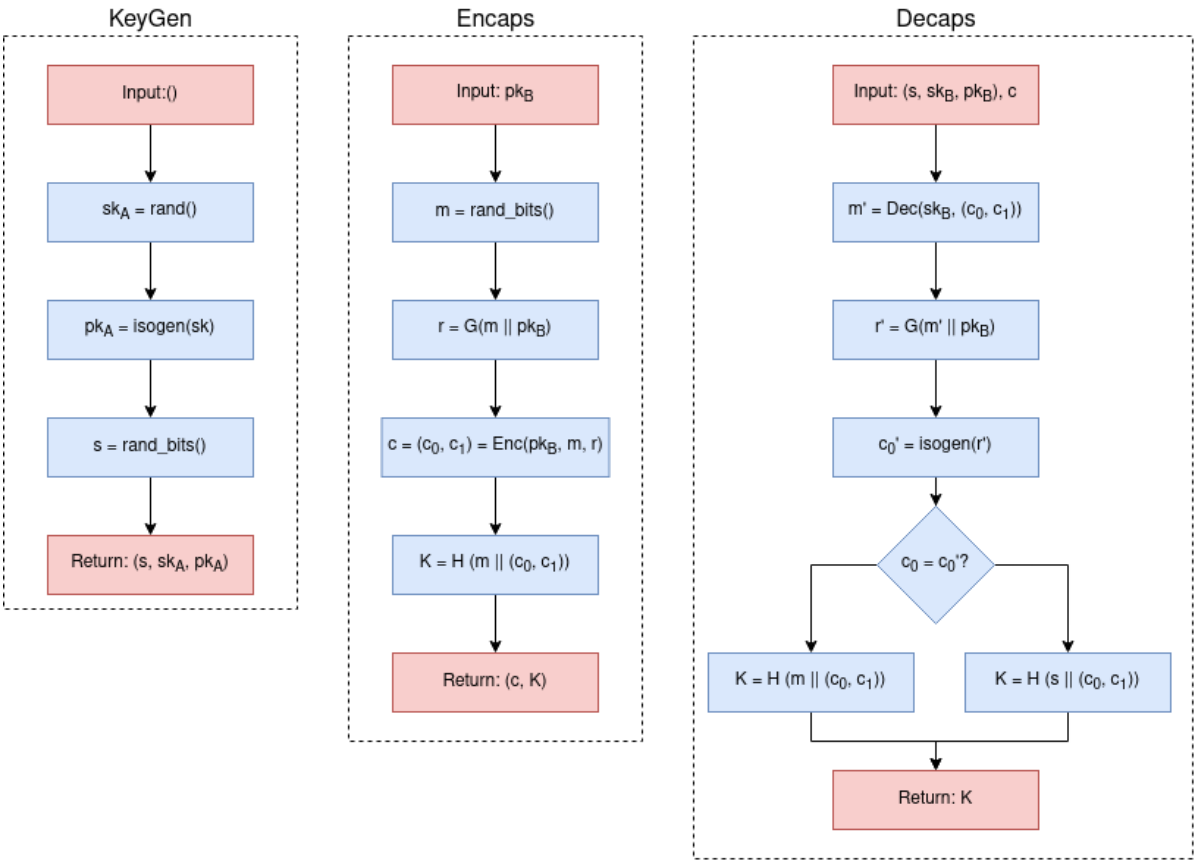

Figure 2.7.: Isogeny-based key encapsulation mechanism (KEM).

2.3.4. Security

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 [27].
The SIKE reference implementation proposes different parameter sets, each supposing to ensure a NIST defined security level. These NIST defined security levels are:

Level 1	Any attack breaking this security level must require resources comparable to perform a key search on a 128-bit key (e.g. AES128).
Level 2	Any attack breaking this security level must require resources comparable to perform a collision search on a 256-bit hash function (e.g. SHA256).
Level 3	Any attack breaking this security level must require resources comparable to perform a key search on a 192-bit key (e.g. AES192).
Level 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).
Level 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 p. This prime p is part of the parameter set itself. See [27] for the detailed definition of each parameter set. Note that the authors of SIKE did not propose a parameter set supposed to satisfy NIST security level 4 (SHA384).

- `p434` supposed to satisfy NIST security level 1 (AES128)
- `p503` supposed to satisfy NIST security level 2 (SHA256)
- `p610` supposed to satisfy NIST security level 3 (AES192)
- `p751` supposed to satisfy NIST security level 5 (AES256)

All isogeny-based cryptographic primitives presented in this chapter can be initialized with one of these parameter sets to match a certain security level. In the context of this thesis the used parameter sets are also called *curves*.

Current research suggest that the SIKE parameter sets satisfy the defined security levels even under the assumption of currently known algorithms [32]. Therefore, the authors consider three algorithms to solve the *SIDH problem*: *Tani's quantum claw finding algorithm* [33], *Grover's algorithm* [12] and a *parallel collision-finding algorithm* [34].
Ephemeral SIDH keys are predestined to implement quantum-secure perfect forward secret protocols [35]. PFS ensures that a compromised long-term key does not reveal past or future keys of the protocol.
Side-channel attacks against isogeny-based 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 implement constant-time cryptography [27]. 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 a challenge and is an active research area.

3. Description of existing SIDH implementations

Currently, three implementations of SIDH are available: *SIKE* [27] (closely related to *PQCrypto-SIDH* [36]), *CIRCL* [37] and *SIKE for Java* [38]. In this chapter each implementation is introduced in detail. At the end of this chapter, Table 3.1 summarizes 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.

3.1. 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 [26]. Today, SIKE is a NIST candidate for quantum-resistant *"Public-key Encryption and Key-establishment Algorithms"*. It is developed by a cooperation of researchers, lead by David Jao [27].

SIKE vs. PQCrypto-SIDH

The SIKE implementation is highly related to the SIDH source code of Microsoft available on Github[1], called PQCrypto-SIDH. At the beginning of this thesis both implementations were considered as different implementations. Since some researches of SIKE also work for Microsoft, the similarities of both APIs seemed to be reasonable. After contacting the SIKE team due to compilation issues of their NIST Round 3 submission, David Jao stated that SIKE uses the *"[G]it repository hosted by Microsoft at https://github.com/Microsoft/PQCrypto-SIDH [...] as the source repository [...] to build the NIST package."* (personal communication, November 17, 2020). This circumstance is unfortunately not documented by SIKE. This communication took place after the major benchmarks of this thesis – include PQCrypto-SIDH – were measured. Thus, the developed benchmarking suite still contains this implementation and the benchmarks are also appended to this thesis. However, this thesis treats both libraries as equal (if not stated otherwise) since both are based on the same source code.

SIKE implements its key encapsulation mechanism (KEM) upon a public key encryption system (PKE) which is built upon SIDH (as described in chapter 2). Besides a generic reference implementation, SIKE offers various optimized implementations of their cryptographic primitives:

[1] `https://github.com/microsoft/PQCrypto-SIDH`

- 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 asserts to countermeasure timing and cache attacks by implementing constant time cryptography [27].

Benchmarked version

The SIKE software release used in this thesis is the official Round 2 submission package[2]. The latest release (Round 3 package) does not support native SIDH compilation as David Jao stated in private communication. Thus, updating to this release is not sufficient. In terms of code optimization the Round 3 package introduces improved algorithms for compressing keys [27]. The differences of these improvements are analyzed using the benchmarks of the above mentioned library PQCrypto-SIDH[3], which contains the improved algorithms.

3.1.1. SIKE API

For all implementations and all parameter sets the API is the same. Therefore, during compilation one need to include the correct subfolders (namely `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`) to initialize SIKE with a specific parameter set.

[2] `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/SIKE-Round2.zip`

[3] Commit: `1f8292d08570fe83c518797b6e103eb8a9f5e6dc`

Key generation

The key generation of the SIKE API consists of the following functions:

Function	Input	Output
`random_mod_order_A`	• Allocated memory for the private key of Alice	• Private key of Alice stored in passed buffer • No return value
`random_mod_order_B`	• Allocated memory for the private key of Bob	• Private key of Bob stored in passed buffer • No return value
`EphemeralKeyGeneration_A`	• Private key of Alice • Allocated memory for the public key of Alice	• Public key of Alice stored in the passed buffer • Returns 0 on success
`EphemeralKeyGeneration_B`	• Private key of Bob • Allocated memory for the public key of Bob	• Public key of Bob stored in the passed buffer • Returns 0 on success

In the following the signatures and high-level workflow of these functions are detailed:

```
// Generate random private key for Alice
void random_mod_order_A(unsigned char* PrivateKey_A);

// Generate random private key for Bob
void random_mod_order_B(unsigned char* PrivateKey_B);
```

Generating a random private key for Alice or Bob. The generated random bytes lie within an interval that is defined by the used parameter set. The underlying randomness source is `/dev/urandom` in Linux and `BCryptGenRandom` in Windows. The generated private key is stored within the pre allocated parameter `PrivateKey_A` or `PrivateKey_B`. No value is returned.

```
// Generate ephemeral public key for Alice
int EphemeralKeyGeneration_A(const unsigned char* PrivateKey_A,
                             unsigned char* PublicKey_A);

// Generate ephemeral public key for Bob
int EphemeralKeyGeneration_B(const unsigned char* PrivateKey_B,
                             unsigned char* PublicKey_B);
```

This function takes the randomly generated private key of Alice (`PrivateKey_A`) or Bob (`PrivateKey_A`) as input and computes the corresponding public key `PrivateKey_A` or `PrivateKey_B`. All compressed versions of SIKE additionally perform a key compression before the generated public key is returned. The function returns 0 if successful.
The function `EphemeralKeyGeneration_X` is an implementation of *isogen* introduced in the previous chapter (see subsection 2.3.3 for details).

Secret generation

The secret generation of the SIKE API consists of the following functions:

Function	Input	Output
`EphemeralSecretAgreement_A`	• Private key of Alice • Public key of Bob • Allocated memory for the secret	• Generated secret stored in passed buffer • Returns 0 on success
`EphemeralSecretAgreement_B`	• Private key of Bob • Public key of Alice • Allocated memory for the secret	• Generated secret stored in passed buffer • Returns 0 on success

In the following the signatures and high-level workflow of these functions are detailed:

```
// Computation of shared secret by Alice
int EphemeralSecretAgreement_A(const unsigned char* PrivateKey_A,
                               const unsigned char* PublicKey_B,
                               unsigned char* SharedSecret_A)

// Computation of shared secret by Bob
int EphemeralSecretAgreement_B(const unsigned char* PrivateKey_B,
                               const unsigned char* PublicKey_A,
                               unsigned char* SharedSecret_B)
```

`EphemeralSecretAgreement_A` uses Alice's `PrivateKey_A` and Bob's `PublicKey_B` to produce a shared secret `SharedSecret_A` between Alice and Bob. All compressed versions of SIKE perform a public key decompression before the shared secret is computed. If successful, this function returns 0. `EphemeralSecretAgreement_B` follows the same principles in order to allow Bob to compute the shared secret.
Both functions are an implementation of *isoex* introduced in the previous chapter (see subsection 2.3.3 for details).

3.2. CIRCL

CIRCL (**C**loudflare **I**nteroperable, **R**eusable **C**ryptographic **L**ibrary) is a collection of cryptographic primitives developed by Cloudflare [37]. CIRCL is written in Go and implements some quantum-secure algorithms like SIDH and an isogeny-based KEM. Note that the GO compiler outputs a native binary. 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
- 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 [39].

Benchmarked version

The software release used in this thesis is identified by the following commit id `0440a499b7237516c7ba535bd1420241e13d385c` of the CIRCL Github repository[4].

3.2.1. CIRCL API

The API of CIRCL is in the following described with respect to functions for key and secret generation (where `XXX` $\in$ {`434, 503, 751`}).

[4] https://github.com/cloudflare/circl/

Key generation

Function	Input	Output
`sidh.NewPrivateKey`	• Parameter set (`p434`, `p503` or `p751`) • Party (Alice or Bob)	• Returns an initialized object of type `PrivateKey`
`PrivateKey.Generate`	• Source of randomness	• Stores random bytes in the `PrivateKey` object • Might return error (e.g. if random number generator fails)
`sidh.NewPublicKey`	• Parameter set (`p434`, `p503` or `p751`) • Party (Alice or Bob)	• Returns an initialized object of type `PublicKey`
`PrivateKey.GeneratePublicKey`	• Object of type `PublicKey`	• The passed object of type `PublicKey` is filled with the public key

```
// 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)
```

The function `NewPrivateKey` returns an object of type `PrivatKey` containing the defined parameter set (`sidh.FpXXX`) and the defined subject (`sidh.KeyVariantSidhA` for Alice and `sidh.KeyVariantSidhB` for Bob). `PrivateKey` object provides the function `PrivateKey.Generate` that randomly generates a new private key based on a passed random number generator (e.g. `rand.Reader`). The generated private key is stored within the attribute `PrivateKey.Scalar` as an array of bytes.
The code above initializes and randomly generates the private keys for Alice (`PrivateKey_A`) and Bob (`PrivateKey_B`).

```
// 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)
```

The function `NewPublicKey` creates an object of type `PublicKey` using the passed parameter set (`sidh.FpXXX`) and the defined subject (`sidh.KeyVariantSidhA` indicates Alice and `sidh.KeyVariantSidhB` indicates Bob). Objects of type `PrivateKey` provide the function `PrivateKey.GeneratePublicKey` that generates the appropriate public key for a given private key. The resulting public key is stored within the `PublicKey` object passed to that function. `PrivateKey.GeneratePublicKey` is an implementation of *isogen* introduced in the previous chapter (see subsection 2.3.3 for details).
The code above firstly initializes the public keys for Alice (`PublicKey_A`) and Bob (`PublicKey_B`). The previously created private keys of Alice and Bob are then used to fill these public key objects.

Secret generation

Function	Input	Output
`PrivateKey.DeriveSecret`	• Allocated memory for shared secret • Object of type `PublicKey`	• Stores the computed secret in the passed memory

```
// 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)
```

In order to create the shared secrets for Alice and Bob, the initially created private key objects provide the function `PrivateKey.DeriveSecret`. This function expects a public key and a previously allocated memory to store the shared secret in. `PrivateKey.DeriveSecret` is an implementation of *isoex* introduced in the previous chapter (see subsection 2.3.3 for details).
The code above firstly allocates the memory for a shared secret and finally generates the shared secret between Alice and Bob.

3.3. SIKE for Java

SIKE for Java implements experimental supersingular isogeny cryptography in Java [38]. The library is developed by *Wutra* in cooperation with *Raiffeisen Bank International*. Besides a slow reference implementation which focuses on code readability, *SIKE for Java* implements an optimized variant that focuses on performance and security. The library implements isogeny based PKE, KEM and SIDH and supports the following parameter sets: `p434`, `p503`, `p610` and `p751`.
Since Java applications run via the Java Virtual Machine (JVM) and are compiled during execution (Just-in-time compiler), the execution of Java programs is significantly slower than executing C or GO programs, which compile the source code before the execution takes place (Ahead-of-time compiler). Thus, comparing SIDH implementations between Java and C/GO does not produce meaningful results. For that reason *SIKE for Java* will not be considered for the benchmarks in this work.

3.3.1. API of SIKE for Java

The API of *SIKE for Java* for a SIDH key exchange is described in the following (`XXX` $\in$ {`434, 503, 610, 751`}).

Initialization

Function	Input	Output
`SikeParamPXXX` (constructor)	• Implementation type (optimized or reference)	• Returns an object of type `SikeParamPXXX` representing the used parameter set
`KeyGenerator` (constructor)	• Parameter object of type `SikeParamPXXX`	• Returns an object of type `KeyGenerator` implementing a key generator
`Sidh` (constructor)	• Parameter object of type `SikeParamPXXX`	• Returns an object of type `Sidh` implementing the secret key computation

```
// Defining parameter set
SikeParam sikeParam = new SikeParamP434(ImplementationType.OPTIMIZED);
```

```
// Initializing key generator
KeyGenerator keyGenerator = new KeyGenerator(sikeParam);

// Prepare SIDH object
Sidh sidh = new Sidh(sikeParam);
```

First of all, the parameter set and the desired implementation is defined by creating an object `sikeParam` of type `SikeParam` (this is a generalization of type `SikeParamXXX`). This object is then used to initialize an object of type `KeyGenerator` (which is later used to generate SIDH keys) and to initialize an object of type `Sidh` (which is later used to compute shared secrets).

Key generation

Function	Input	Output
`KeyGenerator.generateKeyPair`	• Party (Alice or Bob)	• Returns an object of type `KeyPair` containing a random private key and the corresponding public key

```
// Initializing key pair for Alice
KeyPair keyPairA = keyGenerator.generateKeyPair(Party.ALICE);

// Initializing key pair for Bob
KeyPair keyPairB = keyGenerator.generateKeyPair(Party.BOB);
```

The previously generated object `keyGenerator` is used to generate key pairs for Alice and Bob by passing the appropriate party as parameter. Internally this function first generates a random private key. The following computation of the corresponding public key is an implementation of *isogen* described in subsection 2.3.3.
The returned object of type `KeyPair` is later used to access the private and public key.

Secret generation

Function	Input	Output
`Sidh.generateSharedSecret`	• Party (Alice or Bob) • Private key of the chosen party • Public key of the not chosen party	• Returns an object of type `Fp2Element` representing the shared secret

```
// Computation of shared secret by Alice
Fp2Element secretA = sidh.generateSharedSecret(Party.ALICE,
                                               keyPairA.getPrivate(),
                                               keyPairB.getPublic());

// Computation of shared secret by Bob
Fp2Element secretB = sidh.generateSharedSecret(Party.BOB,
                                               keyPairB.getPrivate(),
                                               keyPairA.getPublic());
```

In order to generate the shared secret between Alice and Bob, both parties need to use the previously created `Sidh` object to call `Sidh.generateSharedSecret`. Each party passes the own private key and the public key of the other party in order to compute the shared secret of type `Fp2Element`. This function is an implementation of *isoex* described in subsection 2.3.3.

3.4. Overview

	SIKE	**CIRCL**	**SIKE for Java**
Developer	Research cooperation	Cloudflare	Wultra
Language	C Assembly	GO Assembly	Java
Reference	`www.sike.org`	Github: cloudflare/circl	Github: wultra/sike-java
Implemented primitives	SIDH PKE KEM	SIDH KEM	SIDH PKE KEM
Available parameters	p434 p503 p610 p751	p434 p503 p751	p434 p503 p610 p751
Optimized versions	Generic Generic compressed x64 x64 compressed ARM64 ARM Cortex M4 VHDL	Generic AMD64 ARM64	Generic for Java
Security	Constant time	Constant time	Constant time
Considered for benchmarking	Yes (distinction to PQCrypto-SIDH)	Yes	No (meaningless results due to JVM

Table 3.1.: Overview over existing SIDH implementations.

4. Benchmarking Suite

In this chapter a benchmarking suite is presented that is able to generate comparable benchmarks between all SIDH implementations introduced in chapter 3. To get a better understanding of the results, this chapter starts with a description of the tools and methodologies used for benchmarking (section 4.1). The implementation details given in section 4.2 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 is given in section 4.3.

4.1. 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. The benchmarking suite runs within an Ubuntu Docker container providing a virtual operating system.
The benchmarks measured in this thesis are *execution time* and *memory consumption*. These measurements are used to quantify the claimed resources of an algorithm.
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 3 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, source 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 suite must use consistent compiler optimizations. This ensures the comparability of the binaries. 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 [40]. To be able to extract benchmarks for specific functions *inlining* is disabled during compilation. In order to avoid the modification of the source code of the libraries simple wrapper function were created, which only call the underlying API function. These wrapper function were marked to avoid *inlining*. The C programming language offers the following directive to disable *inlining* for a function:

```
// 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"
```

Compiling the source code determines the Instruction Set Architecture (ISA) of the binary. The architecture of a processor describes the ISA a concrete processor is able to process. When executing the binary the instructions that are defined in the ISA are executed by the processor.

Run *Callgrind*

Callgrind records function calls of a binary. For each call the executed instructions which are processed by the processor are counted. Moreover, the tool provides detailed information about the callee and frequency of function calls. 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 of the benchmarking suite significantly. This is the main reason for the long execution times when collecting benchmarks.

Besides counting the instructions executed by the processor, one could also count elapsed CPU cycles or equivalently the elapsed time for a concrete binary. In the following paragraph the differences are considered and reasons for measuring instructions are given.
Firstly, the measured instruction count of *callgrind* can be converted to elapsed CPU cycles and to thus to the elapsed time. The following calculation exemplarily shows this conversion for the execution of a SIDH key exchange (using the SIKE `x64_optimized` implementation initialized with parameter set `p434`). *Callgrind* measures constantly 64 621 792 instructions when executing that binary. The performance analyzing tool *perf* can also be run on that binary (`sudo run perf` *`binary`*). Besides the elapsed CPU cycles and elapsed time *perf* also measures the executed instructions per CPU cycle (IPC) and the average CPU frequency in GHz. This values can be used to compute the elapsed time of an application. It is important to note, that the IPC is not a characteristic index for a CPU, moreover it varies between multiple runs of the same application based on scheduling decisions and hardware dependencies (e.g. cache size) [41]. Thus, the measured CPU cycles and the corresponding elapsed time of an application is not constant. The measured IPC for the binary (SIKE `x64_optimized` using `p434`) is on average 2.87 (10 samples). The measured CPU frequency is on average 2.66 *GHz* (10 samples). Thus, the elapsed time depending on the measured instructions can be computed as follows [41]:

$$Elapsed\ Time = \frac{Instructions}{Instructions\ per\ Cycle\ (IPC) \cdot CPU\ Frequency\ in\ GHz \cdot 10^9}\ [s]$$

$$= \frac{Instructions}{2.87 \cdot 2.66 \cdot 10^9}\ [s] = \frac{Instructions}{7,6342 \cdot 10^9}\ [s]$$

For the instructions counted using *callgrind* this formula yields:

$$Elapsed\ Time\ = \frac{Instructions}{7,6342 \cdot 10^9} = \frac{64\,621\,792}{7,6342 \cdot 10^9} = 0.0084647759\ [s] = 8.465\ [ms]$$

At the same time the measured execution time by *perf* is on average 8.476 $[ms]$ (10 samples). Since the difference is marginal it seems to be sufficient to provide the instruction counts instead of measuring the exact execution times.
Secondly, the elapsed time for an application depends on the IPC, which is a highly dynamic value (e.g. depending on scheduling decisions, cache misses or parallelism of the underlying CPU [41]). At the same time *callgrind* measures the executed instructions of the compiled application, which is a constant value if the binary itself implements static runtime behavior (e.g. the measured instructions of a binary printing *"Hello World"* once is constant, while

the executed instructions of a binary printing a randomly chosen amount of *"Hello World"* messages is not constant). Thus, the measured instructions only depend on the compiler and the CPU architecture (ISA) and they are independent of the dynamic behavior of the concrete underlying CPU. Besides abstracting from the concrete CPU this also increases reproducibility of the benchmarking results.

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 parse and 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 measured multiple times (e.g. $N = 100$ times) and averaged over all outcomes.

4.2. 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 and 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 and 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 and p751`
  - CIRCL generic optimized implementation (`CIRCL_Generic`)
  - CIRCL x64 optimized implementation (`CIRCL_x64`)

Furthermore, the benchmarking suite provides ECDH implemented in *OpenSSL*. Since SIDH is a candidate to replace current Diffie-Hellman algorithms, ECDH is intended as reference value: The objective is to compare optimized state-of-the-art ECDH with quantum-secure SIDH.
Since each parameter set of SIDH meets a different security level (compare subsection 2.3.4) ECDH is also instantiated with different curves, each matching a SIDH security level. The used elliptic curves are:

1. `secp256r1` also known as `P-256` (*OpenSSL* implementation: `prime256v1` [42]):
   128 bit security matching `p434` [43]
2. `secp384r1` also known as `P-384` (*OpenSSL* implementation `secp384r1`):
   192 bit security matching `p610` [43]
3. `secp521r1` also known as `P-521` (*OpenSSL* implementation `secp521r1`):
   256 bit security matching `p751` [43]

Note that the classical security assumptions for `p434` (key search for 128 bit AES) and `p503` (collision search for 256 bit SHA) are comparable: A key search for a 128 bit AES key takes 2^{128} operations indicating a security level of 128 bits [44]. Due to the birthday attack [45], the collision finding on a 256 bit hash function requires about $2^{256/2} = 2^{128}$ operations leading security level of 128 bit [44]. Thus, both parameter sets `p434` and `p503` provide similar classical security levels. NIST maps this security level to elliptic curves employing key sizes of 256 bit (e.g. curve `secp256r1`). Hence, no separate elliptic curves for `p434` and `p503` are included. For completion: Security levels of 192 bit and 256 bit are mapped to elliptic curves providing of key sizes 384 bit (e.g. `secp384r1`) and 512 bit (e.g. `secp512r1`), respectively [46].
For each of these introduced implementations the application measures benchmarks. The following sections present detailed information about the internals of the suite. Besides the precise application flow (subsection 4.2.1), the internal class structure of the Python3 code is shown (subsection 4.2.2).

4.2.1. Application Flow

This sections illustrates the application flow of the benchmarking suite (see Figure 4.1). 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*. Outputs of *Callgrind* and *Massif* are parsed and saved in the benchmarking suite after each run.
Finally, the results are visualized in different formats. Graphs compare the recorded instruction counts and the peak memory consumption among implementations instantiated with comparable security classes. The HTML and LaTeX tables lists detailed benchmarks per implementation.

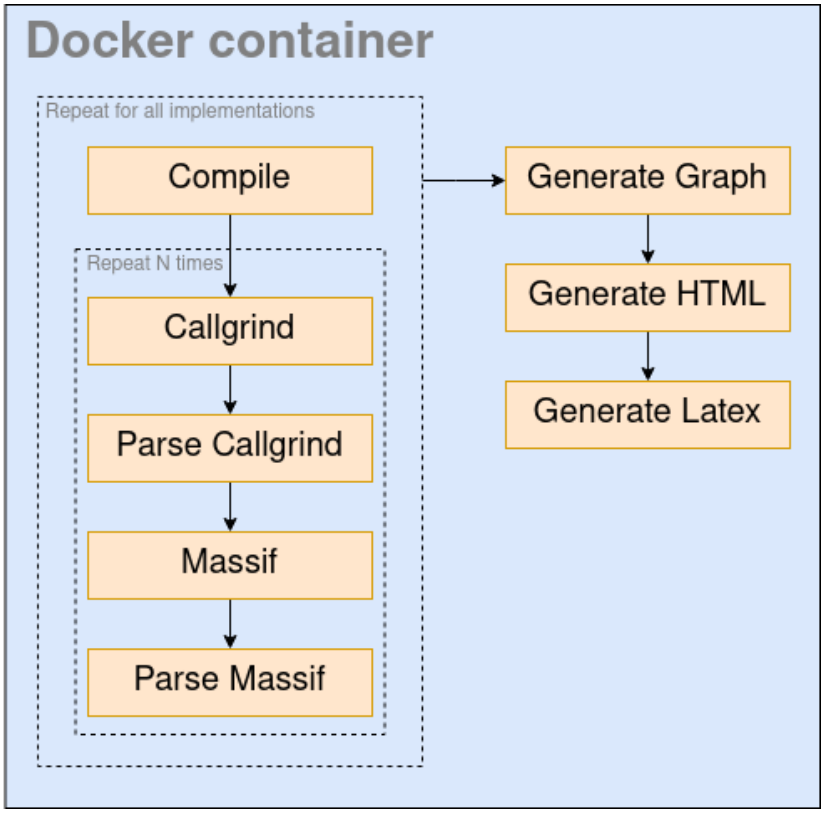

Figure 4.1.: Flow chart of the benchmarking suite. For each implementation, the source code is compiled and benchmarked multiple times. The benchmarking results are visualized in different format: Graphs, HTML and LaTeX.

4.2.2. 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, consider the highly-documented

source code of the benchmarking suite.

Representation of concrete implementations

The main logic of the benchmarking suite is placed within the class `BaseImplementation`. This class implements the logic for compiling, executing *Callgrind* and *Massif* and parsing their results. For each implementation which shall be benchmarked, a subclass of `BaseImplementation` is created (see Figure 4.2). These subclasses provide a link to the respective *Makefile* which is used by the `BaseImplementation` for compiling, running callgrind and running massif. Furthermore, each subclass can provide a set of arguments passed to the *Makefile*.

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 hierarchy shown in Figure 4.3 is applied.
The class `BenchmarkImpl` 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.

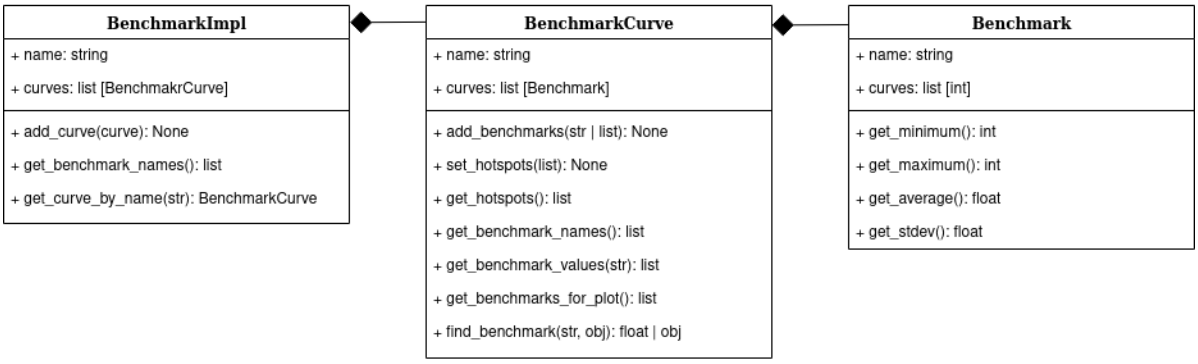

Figure 4.3.: Class diagram representing the management of the benchmarking results.

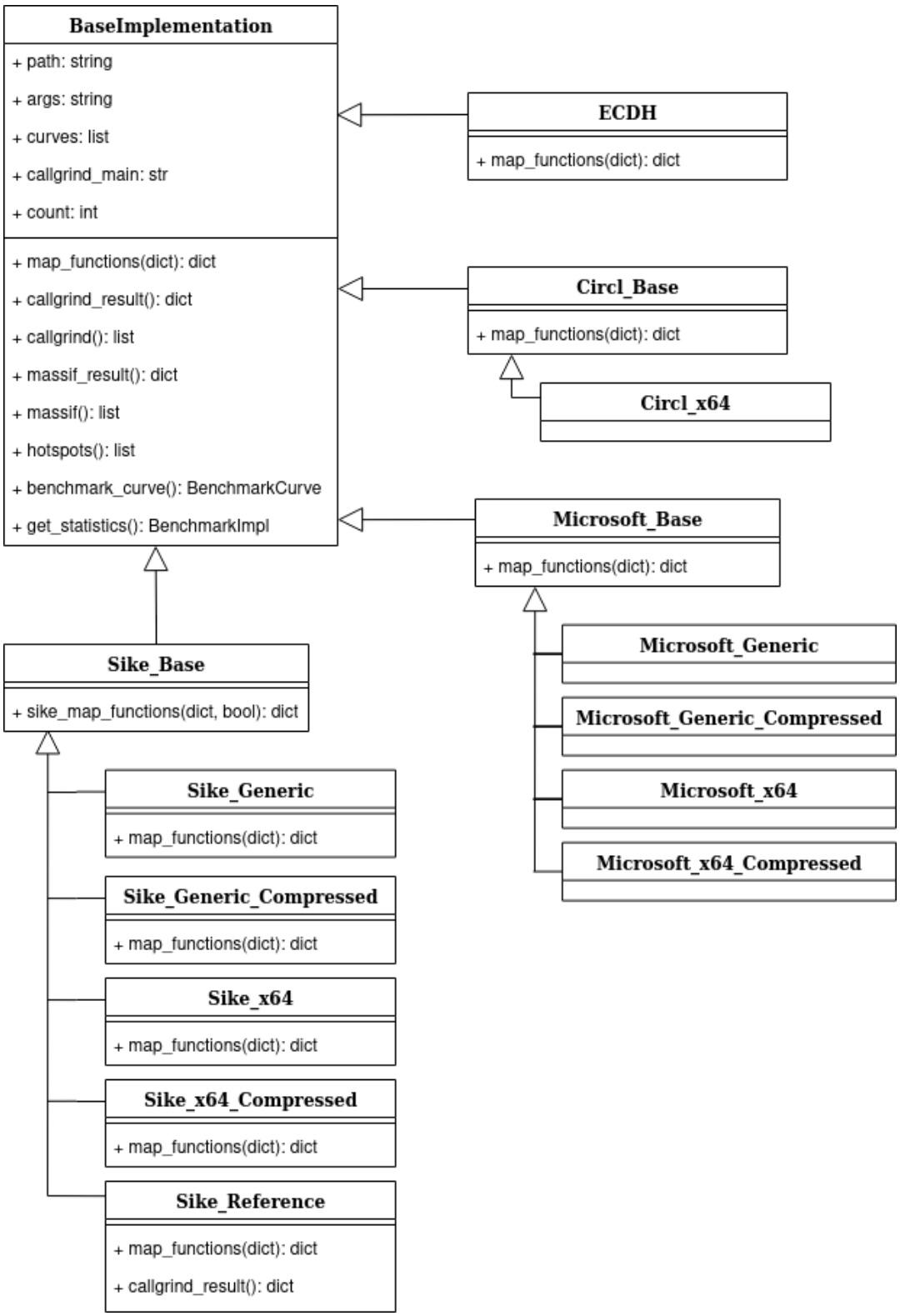

Figure 4.2.: Class diagram for all implementations supported by the benchmarking suite. All concrete implementations of SIDH inherit from `BaseImplementation`. A *Makefile* provided by each subclass specifies how the benchmarking code is built and run.

4.2.3. Adding Implementations

To add a new implementation to the benchmarking suite it is recommended to consider already existing implementations. Moreover, appendix A.1 might be helpful to get a better understanding about the applied project hierarchy. 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` supporting 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:

```
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:

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

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

4.3. Usage

This section explains the usage of the benchmarking suite. Besides the execution of benchmarks, the application also provides unit tests for the suite itself. 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 an easy to use interface for both tasks a script *run.sh* is available (see addenda A.1 for the project hierarchy). This script can be invoked 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). The frequency of measurements for each benchmark is configurable: 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 `data/` folder of

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

your current directory. These files visualize average values over N samples. The output files are:

- *XXX.png*: These files compare the absolute instruction counts 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 *json* file contains cached benchmarking results. It can be used as input for the benchmarking suite to import 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 speeds up the generation of the output graphs significantly. 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.
- *results.tex*: This file contains the benchmarking results formatted in LaTeX. The output is used for this thesis.

5. Benchmarking Results

The results presented in this chapter have been 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. $N = 100$). All values presented in this chapter are averages over $N = 100$ samples. Additionally, all graphs contain the standard deviation (if no standard deviation can be observed in a particular graph, the value is zero or too small to be drawn). The exact values and the identified execution hotspots for all implementations instantiated respectively with all parameter sets (see section 4.2) can be found in addenda A.2. The efficiency in this chapter is quantified by the peak memory consumption and by the absolute instruction count measured by the benchmarking suite (details in chapter 4).
The graphs below 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.

All graphs in this chapter also contain ECDH benchmarks as reference value. However, mainly section 5.4 will investigate and analyze these differences.

This chapter starts with a comparison between all SIDH security levels in section 5.1. This demonstrates the performance differences of the available parameter sets.
A comparison among all available SIKE implementations is given in section 5.2: This section works out differences between *reference*, *generic optimized*, *x_64 optimized* and *compressed* implementations of SIKE. Moreover, the previously mentioned similarities and differences between PQCrypto-SIDH and SIKE are investigated based on the measured benchmarking results.
In section 5.3 the comparison between SIKE and CIRCL is drawn to identify the most performant SIDH key exchange implementation. This section also analyses the observed execution hotspots of both libraries.
Differences in terms of efficiency between modern state-of-the-art ECDH and quantum-resistant SIDH are pointed out in section 5.4. This section also highlights execution hotspots measured for ECDH.
The chapter concludes with security considerations for all benchmarked implementations in

terms of constant time cryptography and key size (section 5.5).
At the end of each section a short summary of the results for the respective section can be found.

5.1. Comparing of SIDH security levels

Before drawing any differences between libraries or implementations, this section considers the proposed parameter sets for isogeny based cryptography [27]: `p434`, `p503`, `p610` and `p751`. Each parameter set corresponds with a NIST security level (see subsection 2.3.4 for details). This section visualizes the claimed resources of the different parameter sets for a single SIDH key exchange. SIKE (subsection 5.1.1) and CIRCL (subsection 5.1.2) are investigated separately. Additionally, this section also compares the proposed SIDH parameters to classical elliptic curve cryptography parameters: `secp256r1`, `secp384r1` and `secp521r1`. Recall the security classes of all parameter sets (details in section 4.2):

Security Level	SIDH parameters	ECDH parameters
NIST level 1	`p434`	`secp256r1`
NIST level 2	`p503`	-
NIST level 3	`p610`	`secp384r1`
NIST level 5	`p751`	`secp521r1`

Table 5.1.: NIST security levels with SIDH and ECDH parameters.

This section concludes with a short summary of all identified findings (subsection 5.1.3).

5.1.1. Security levels of SIKE

This section evaluates the performance of the different parameter sets implemented in the SIKE library. SIKE supports all proposed parameter sets: `p434`, `p503`, `p610` and `p751`. The comparison is based on the `SIKE_x64` implementation.

Figure 5.1 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 observed regarding *KeygenB*, *SecretA* and *SecretB*. The other parameter sets `p503` and `p610` almost lie on a linear line between the highest and lowest security class. At the same time the ECDH implementations are clearly faster while providing the same security levels: `Secp256r1` processes in total 1.7 million instructions while `p434` executes 69 million operations (40.6 times more) for a single SIDH key exchange. `Secp384r1` (40 million) executes 4.3 times more instructions than `p610` (170 million) and `secp521r1` (96 million) executes 2.6 times more instructions than `p751` (253 million).
The measured peak memory consumption is for `p751` the highest (13.3 kB) and for `p434` the lowest (8.2 kB) among SIDH parameters. Again, the other parameter sets can be found in

between of these boundary values (Figure 5.2). Moreover, one can observe an increased memory consumption of ECDH parameters: Compared to `p434`, the elliptic curve parameter set `secp256r1` demands 4 *kB* more memory. `Secp384r1` and `secp521r1` allocate 3 *kB* additional memory compared to their appropriate SIDH parameters, respectively.

This analysis clearly demonstrates that an increased security level of SIDH corresponds with an increased claim of resources. Whereas the difference in terms of memory consumption is relatively small, the execution times differ noticeably: On average and compared to `p434` the parameter set

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

While the ECDH parameter sets enable faster computation times for a single key exchange, they also suffer from an increased memory consumption compared to the SIKE implementations. The reason for this might be a trade-off between execution time and memory consumption. This is further investigated in section 5.4.

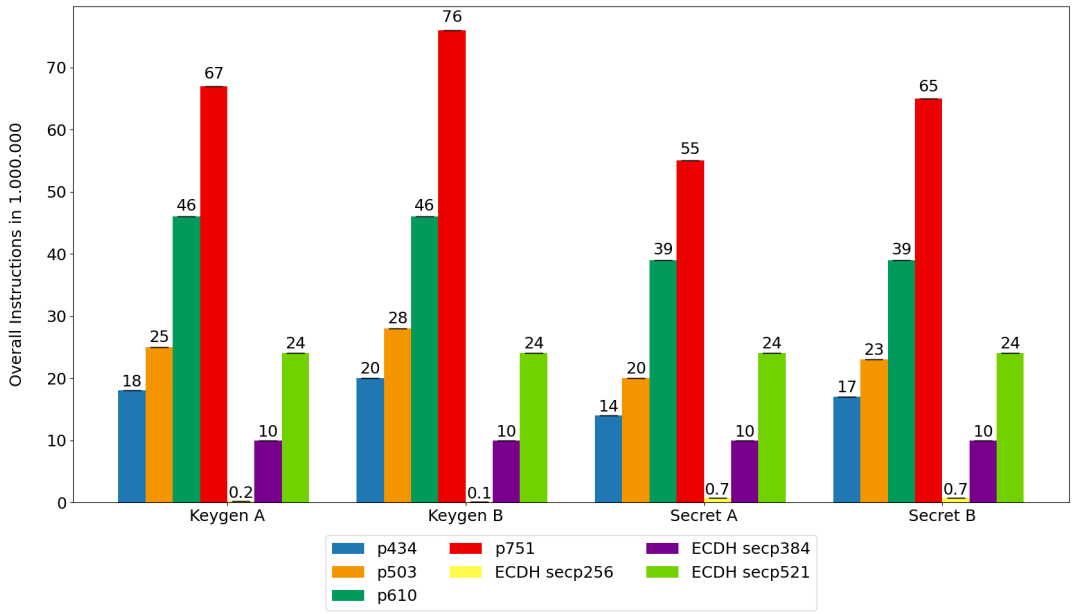

Figure 5.1.: Overall instructions for `SIKE_x64` initiated with all possible parameter sets compared to appropriate `ECDH` parameter sets.

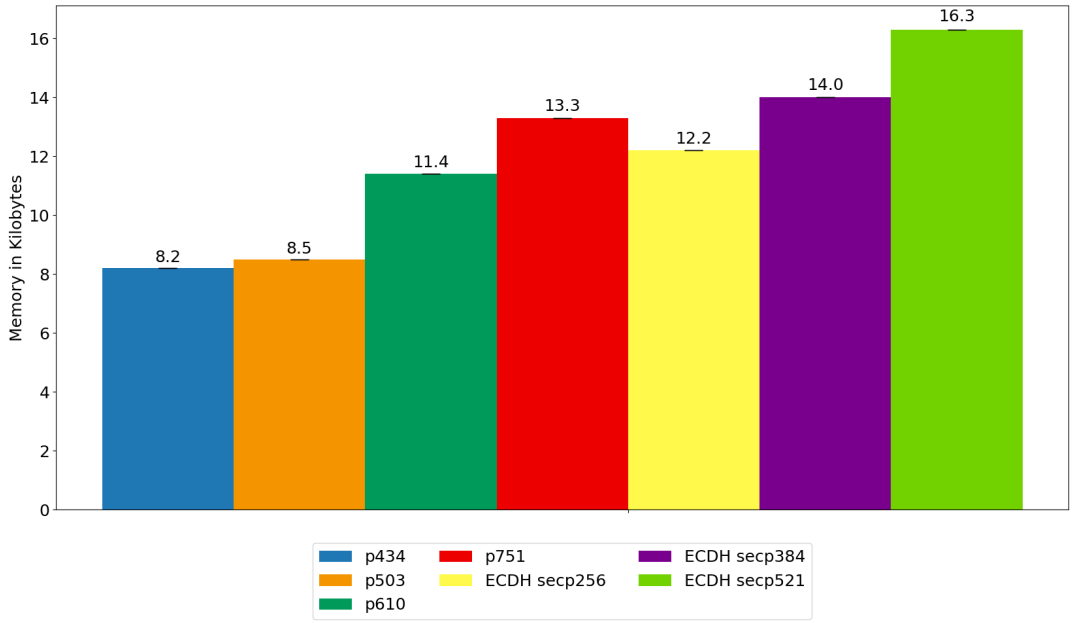

Figure 5.2.: Maximum memory consumption in kilobytes of `SIKE_x64` initiated with all possible parameter sets compared to appropriate `ECDH` parameter sets.

5.1.2. Security levels of CIRCL

This section evaluates the performance of the different parameter sets implemented in the CIRCL library. CIRCL supports the following parameter sets: `p434`, `p503`, and `p751`. The comparison is based on the `CIRCL_x64` implementation.

Figure 5.3 visualizes the executed instructions of all implemented parameter sets in CIRCL. Additionally, the graph also shows the benchmarks for ECDH and the appropriate parameters (`p434` matching `secp256r1` and `p751` matching `secp521r1`). As expected, `p434` is the fastest implementation among the SIDH parameter set (this holds for all categories). In total a whole key exchange using parameter `p434` processes 101 million instructions, while parameter `p503` (114 million) is slightly slower and parameter `p751` (359 million) is by far the slowest version. The overhead of processed instructions for `p751` compared to the other parameter sets is significant. When comparing CIRCL to ECDH a significant speed difference can be observed: `Secp256r1` executes in total 1.7 million instructions (95 times less than `p434`) and texttttSecp521r1 executes in total 96 million instructions (4 times less than `p751`). Thus, ECDH is clearly faster for both parameter sets.
The analysis of the allocated memory of the different parameter sets reveals a dynamic allocation behavior of the CIRCL library (Figure 5.4). For each parameter set the peak memory consumption varies between multiple measurements (see the standard deviation in the graph). The claimed memory for the different parameter sets is almost equal: `p434` allocates on average about 22 kB memory, `p503` about 21.6 kB and `p751` about 23.1 kB. At the same time

CIRCL allocates more memory than ECDH: `Secp256r1` uses 12.2 kB memory and `Secp521r1` allocates 16.3 kB.

Increased security levels of SIDH corresponds with an increased amount of executed instructions. The remarkable increase between `p434` and `p710`, however, can also be observed for the SIKE library (see subsection 5.1.1). Compared to `p434` the parameter set

- `p503` executes on average 1.1 times more instructions
- `p710` executes on average 3.6 times more instructions

The observed dynamic memory allocation behavior of CIRCL might be a consequence of the garbage collector implemented in the Go language [47]. The GO garbage collector starts in parallel with the main executable on multi processor architectures [48]. Due to the state of the CPU during execution and due to the dynamic cache behavior, the garbage collector could free memory in a non-deterministic manner. This might lead to a different peak memory consumption during multiple runs of the binary.
The observed memory allocations by CIRCL marginally differ for different parameter sets. Again, an explanation for this behavior could be the GO garbage collector. The garbage collector knows an upper bound of memory that might be allocated by a binary. Once this upper bound is reached the garbage collector gets active and frees unused memory [47]. This circumstance justifies the constant memory consumption of all SIKE parameter sets.

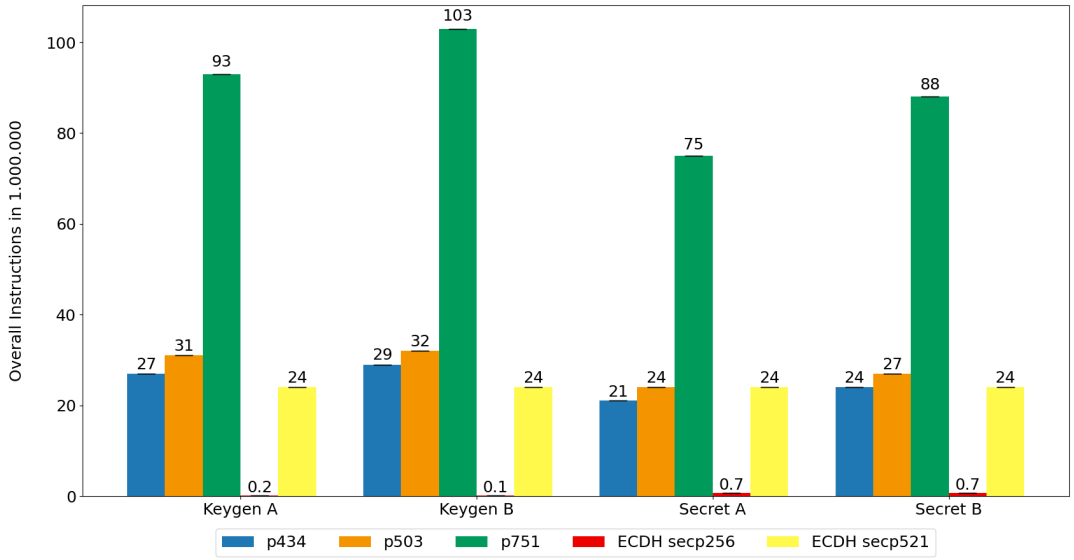

Figure 5.3.: Overall instructions for `CIRCL_x64` initiated with all possible parameter sets compared to appropriate `ECDH` parameter sets.

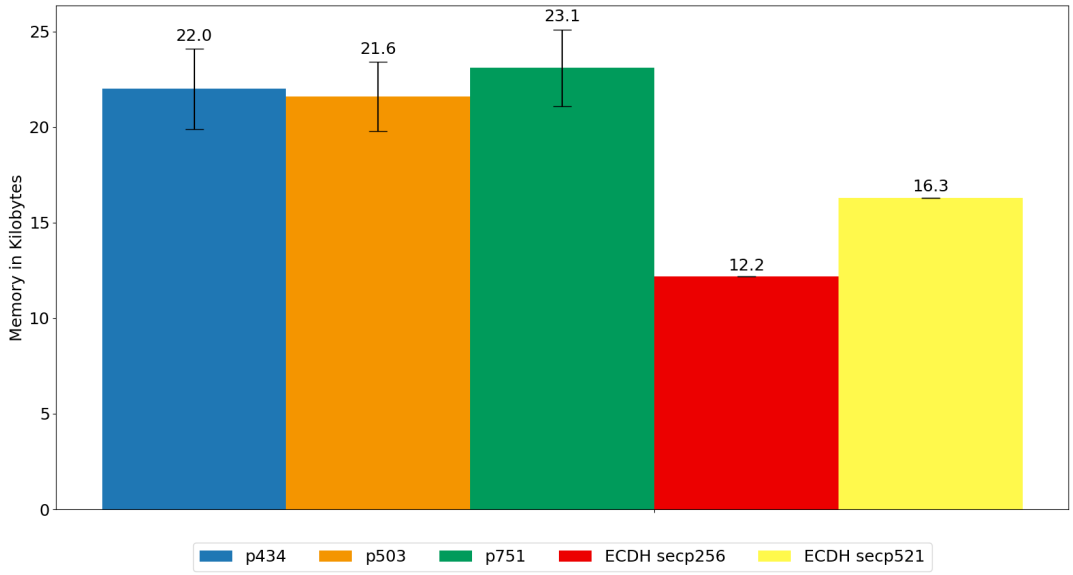

Figure 5.4.: Maximum memory consumption in kilobytes of `CIRCL_x64` initiated with all possible parameter sets compared to appropriate `ECDH` parameter sets.

5.1.3. Summary of findings

This section reveals that an increased level of security does not necessarily correspond with an linear increase of resources (executed instructions and allocated memory). While this assumption holds for the SIKE library, the GO runtime (e.g. the GO garbage collector) affects the memory benchmarks for the CIRCL library. Thus, the chosen programming language influences the claimed resources of SIDH security parameters.
ECDH is faster for all security parameters than SIKE and CIRCL. However, SIKE requires less memory than ECDH.

5.2. Comparing SIKE implementations

Before the comparisons between the SIDH libraries *SIKE* and *CIRCL* are drawn, the differences of available *SIKE* implementations are investigated. The relation between SIKE and PQCrypto-SIDH is evaluated in subsection 5.2.1. Finally all findings are shortly summarized in subsection 5.2.2.
Besides a *reference* implementation, the SIKE library provides *generic optimized* and *x64 optimized* variants. This section analyzes the differences between these *optimized* implementations. Moreover, *compressed* versions for both *optimized* variants are available. The performance differences between *compressed* and *non-compressed* versions are additionally emphasized in this section. Since the previous section investigated differences between SIDH parameter sets, this section only considers `p434`. The analysis of detailed benchmarks listed in addenda A.2 for other parameter sets (`p503`, `p610` and `p751`) lead to similar results as presented in this

section.

Reference implementation

The `SIKE_Reference` implementation is with 11.2 kB allocated memory close to the average ($\sim$ 12.7 kB) of all SIKE implementations (see Figure 5.6). Regarding the execution times for key generation in Figure 5.5, the reference implementation is by far the slowest variant: 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, performance criteria are no requirements for this implementation, which is confirmed by the observed benchmarking results.

Optimized implementations

SIKE provides two *optimized* implementations: `SIKE_Generic` and `SIKE_x64`. `SIKE_x64` is optimized for the x64 instruction set taking advantage of architecture specific operations. `SIKE_Generic` implements generic code without optimizing code for a concrete architecture. Both variants are clearly faster than the reference implementation meeting intuitive expectations (Figure 5.5). The benchmarks in Figure 5.5 show a significant speed up of `SIKE_x64` compared to `SIKE_Generic`. Key generation for Alice and Bob is 10.8 times faster and secret generation about 11 times. At the same time `SIKE_Generic` allocates 7.7 kB memory, which is 0.5 kB less than the *x64 optimized* variant (Figure 5.6). Thus, the increased speed involves the allocation of more memory – a time-memory trade-off (e.g. [49]). At the same time, the executed instructions show the advantage of optimizing SIDH for a concrete hardware architecture instead of providing generic implementations.

Compressed implementations

SIKE provides *compressed* variants for both optimized implementations:
`SIKE_Generic_Compressed` and `SIKE_x64_Compressed`. Both *compressed* versions also contain *optimized* code. Their key sizes, however, are reduced compared to *non-compressed* variants. In order to reduce the key sizes these algorithms implement functions to compress and to decompress cryptographic keys.

Figure 5.5 clearly shows a difference between *optimized* and *compressed* versions of SIKE in terms of absolute instructions. *Compressed* versions roughly claim about twice as much operations ($\sim$ 500 million for `SIKE_Generic_Compressed` and $\sim$ 45 million `SIKE_x64_Compressed`) to generate a key pair as *non-compressed* variants ($\sim$ 200 million for `SIKE_Generic` and $\sim$ 19 million `SIKE_x64`). Additionally, the generation of the shared secret using *compressed* implementations is slightly slower. For all *compressed* versions the executed instructions vary (see the standard deviation) because the compression of the public keys depends on a randomly generated key, which is different for each SIDH key exchange.

Besides execution times, Figure 5.6 also compares the memory consumption of *compressed* implementations. As stated by the authors of SIKE, *compressed* variants allocate more memory [27]: The peak memory allocation is with 17 kilobytes (`SIKE_Generic_Compressed`) and 19.2 kilobytes (`SIKE_x64_Compressed`) twice as high as for the *non-compressed* versions. The measured peak memory consumption is constant for *compressed* implementations and does not vary.

This shows that *optimized* versions have reduced execution times as well as decreased memory consumption compared to *compressed* implementations. Roughly speaking, *compressed* versions demand twice as much resources than *optimized* variants, while decreasing the effective key sizes (see subsection 5.5.2 for details). Since the key compression and decompression force additional steps during a SIDH key exchange, the increased claim of resources is reasonable.

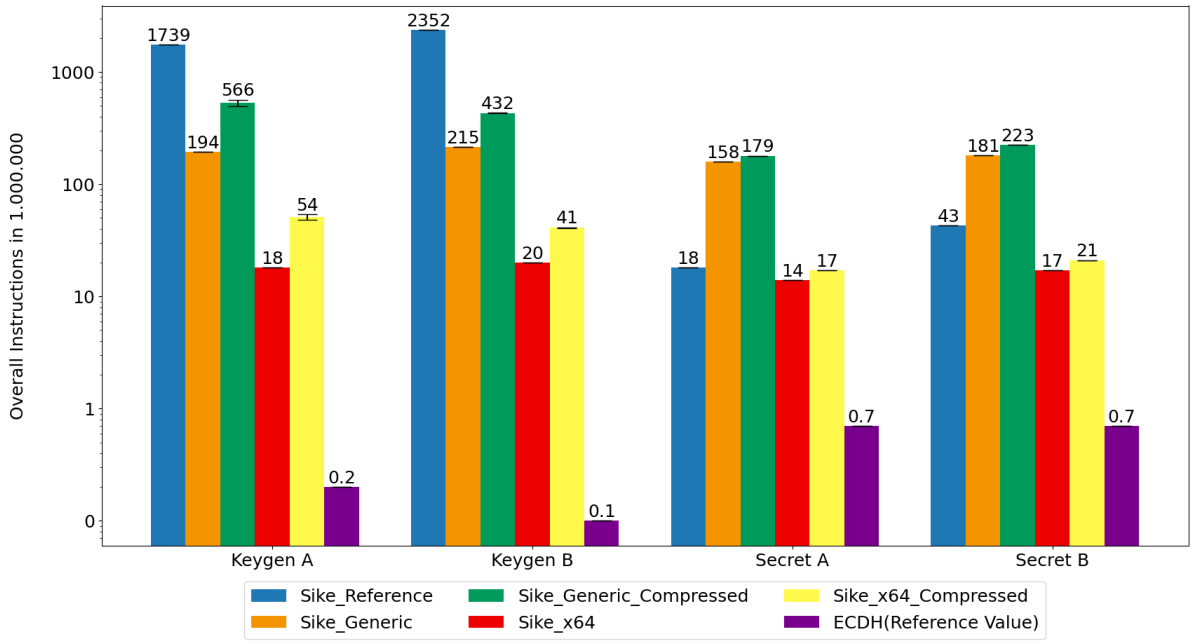

Figure 5.5.: Overall instructions for all SIKE implementations initialized with `p434`. ECDH is shown as reference value using parameter `secp256r1`.

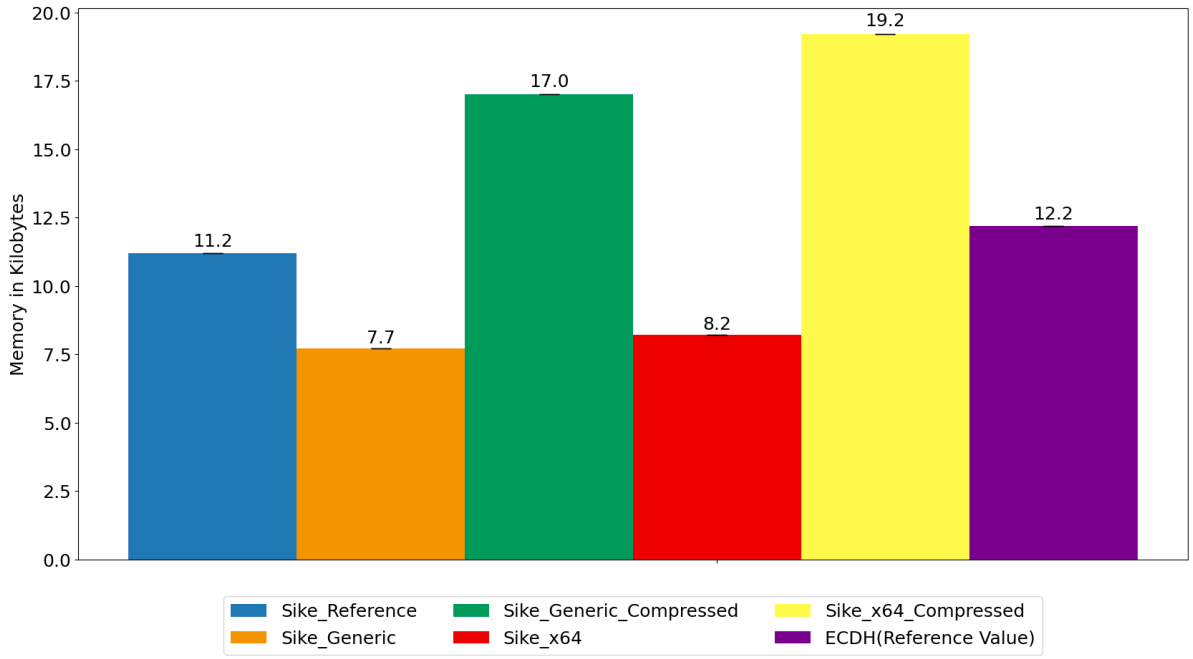

Figure 5.6.: Maximum memory consumption in kilobytes or all SIKE implementations initialized with `p434`. ECDH is shown as reference value using parameter `secp256r1`.

5.2.1. PQCrypto-SIDH vs. SIKE

This section describes the similarities and differences between PQCrypto-SIDH and SIKE. Note, that the SIKE team uses the Microsoft Github repository of PQCrypto-SIDH to create their NIST submissions (see section 3.1 for details). This similarity can be observed when analyzing the *optimized* implementations. At the same time the benchmarked versions of the libraries reveal considerable differences when analyzing their *compressed* versions. This is investigated in the following subsections.

Optimized implementations

This section compares the *optimized* implementations of SIKE and PQCrypto-SIDH: `SIKE_Generic`, `SIKE_x64`, `Microsoft_Generic` and `Microsoft_x64`. Generic optimized implementations contain generally optimized code for various hardware platforms. In contrast, x64 optimized variants exploit hardware specific instructions.

Figure 5.7 shows the benchmarks for execution time of the optimized variants initiated with `p434`. In all four steps of the SIDH key exchange, the benchmarked values for both libraries are similar. However, when considering exact measurements from A.2, 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, when higher security classes are analyzed (about three million operations constant difference for `p751`), the relative disparity stays below 0.01%. Similar results can be observed for *SIKE_x64* and *Microsoft_x64*. There are almost no differences between the two libraries for *optimized* implementations considering the counted instructions.
Peak memory consumptions for parameter *p434* are visualized in Figure 5.8. As in terms of execution time, memory allocation numbers between *SIKE_Generic* and *Microsoft_Generic* marginally differ: The SIKE version occupies 0.3 kB less memory for *p434* and 0.6 kB less than for *p751*. Overall, the relative difference regarding memory consumption is about 5%. Again, the same is valid for *SIKE_x64* and *Microsoft_x64*: The Microsoft implementation allocates 0.7 kB more memory for *p434* and 1.7 kB more than for *p751*.

Both implementations hardly differ in their performance benchmarks. This analysis reveals the similarities of the libraries SIKE and PQCrypto-SIDH. The fact that both libraries relay on the same software stack is mirrored in the benchmark results. The slight performance differences might be caused by the different software releases used for benchmarking (see section 3.1 for details):

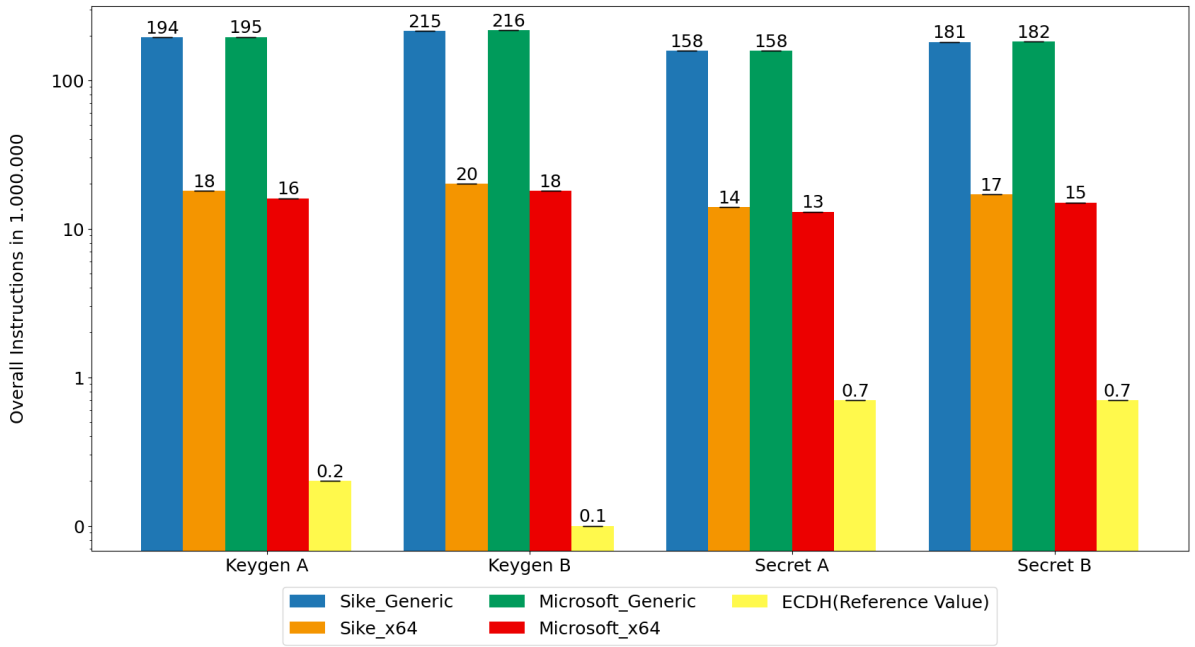

Figure 5.7.: Overall instructions for optimized implementations of SIKE and PQCrypto-SIDH using `p434` compared to ECDH via `secp256r1`.

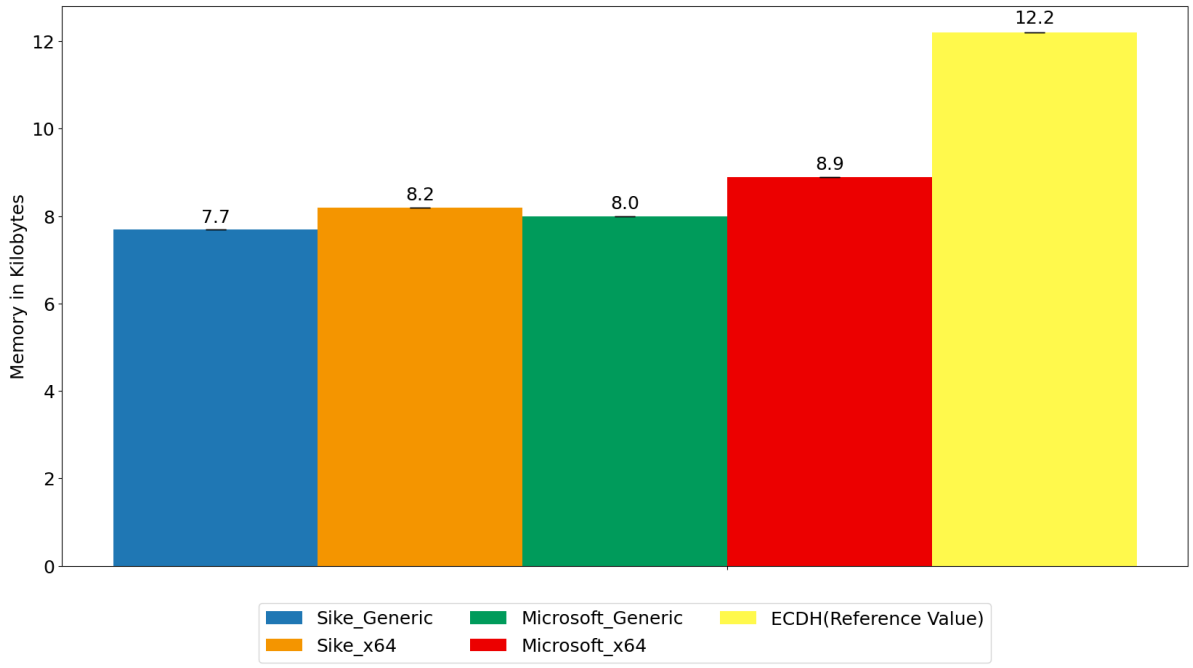

Figure 5.8.: Maximum memory consumption in kilobytes for optimized implementations of SIKE and PQCrypto-SIDH using `p434` compared to ECDH via `secp256r1`.

Compressed implementations

Recall that compressed implementations of SIDH promise shortened key size compared to non-compressed versions, however, this leads to increased execution times and memory allocations. As mentioned above the benchmarked SIKE version (Round 2 submission) does not implement the latest optimizations for compressed versions. However, the analyzed version of PQCrypto-SIDH already integrates these improvements. Thus, this section reveals the differences between the benchmarked versions of SIKE and PQCrypto-SIDH, which is in particular the integration of faster compressing algorithms [50].

Figure 5.9 shows the executed instructions for the compressed variants of PQCrypto-SIDH and SIKE initiated with `p434`. Naturally, the x64 optimized code is faster than the generic optimization. The Microsoft implementations executed less operations than SIKE: Regarding key generation, *SIKE_Generic_Compressed* performed on average 38% more instructions than *Microsoft_Generic_Compressed*. *SIKE_x64_Compressed* performed on average 45% more instructions than *Microsoft_x64_Compressed*. The generation of the secret key only shows slight differences between SIKE and Microsoft, however Microsoft remains faster. The trend of this analysis also applies to improved security classes, e.g to parameter *p751* (see Figure 5.11). Again, these compressed algorithms vary in terms of executed instructions (see the standard deviation), since the effort for compression depend on a randomly chosen key.
The following evaluation of the allocated memory is surprising (Figure 5.10): 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 5.12), where the *PQCrypto-SIDH* library of Microsoft allocates almost seven times more memory.

While the benchmarks for the compressed Microsoft implementations reveal faster execution times, the overhead of allocated memory compared to SIKE is enormous. The observed differences are the results of the integration of optimized compression algorithms [50]: Reducing the execution times for compression directly increases the memory consumption of the algorithms – a time-memory trade-off.

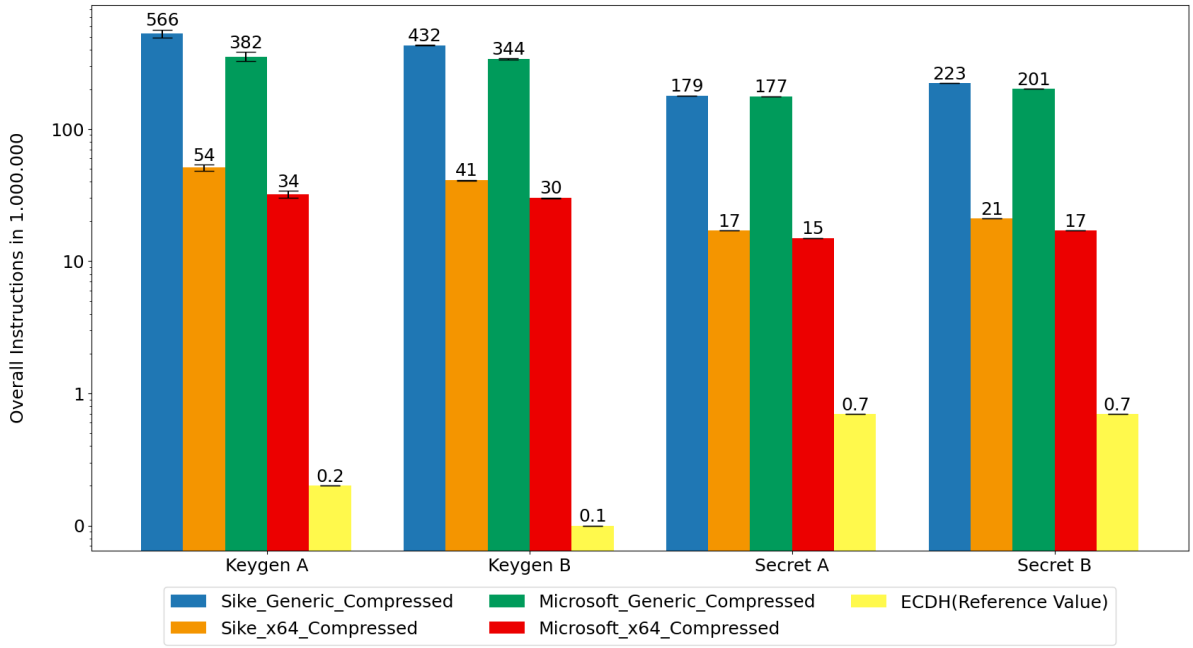

Figure 5.9.: Overall instructions for compressed implementations of SIKE and PQCrypto-SIDH using `p434` compared to ECDH via `secp256r1`.

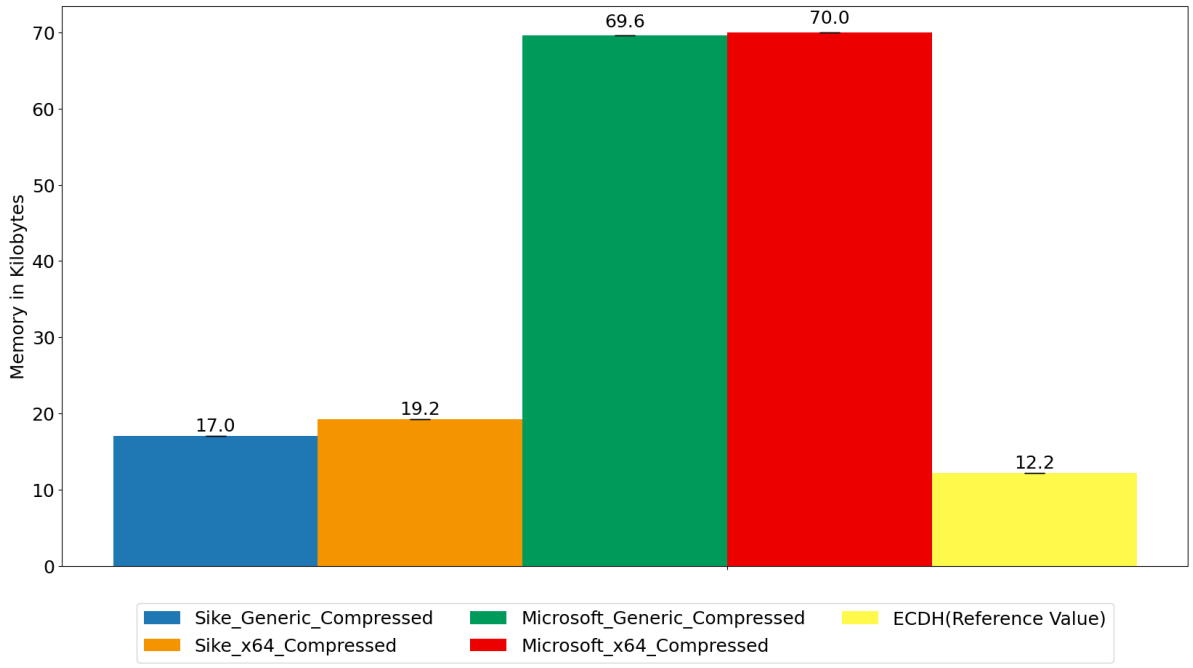

Figure 5.10.: Maximum memory consumption in kilobytes for compressed implementations of SIKE and PQCrypto-SIDH using `p434` compared to ECDH via `secp256r1`.

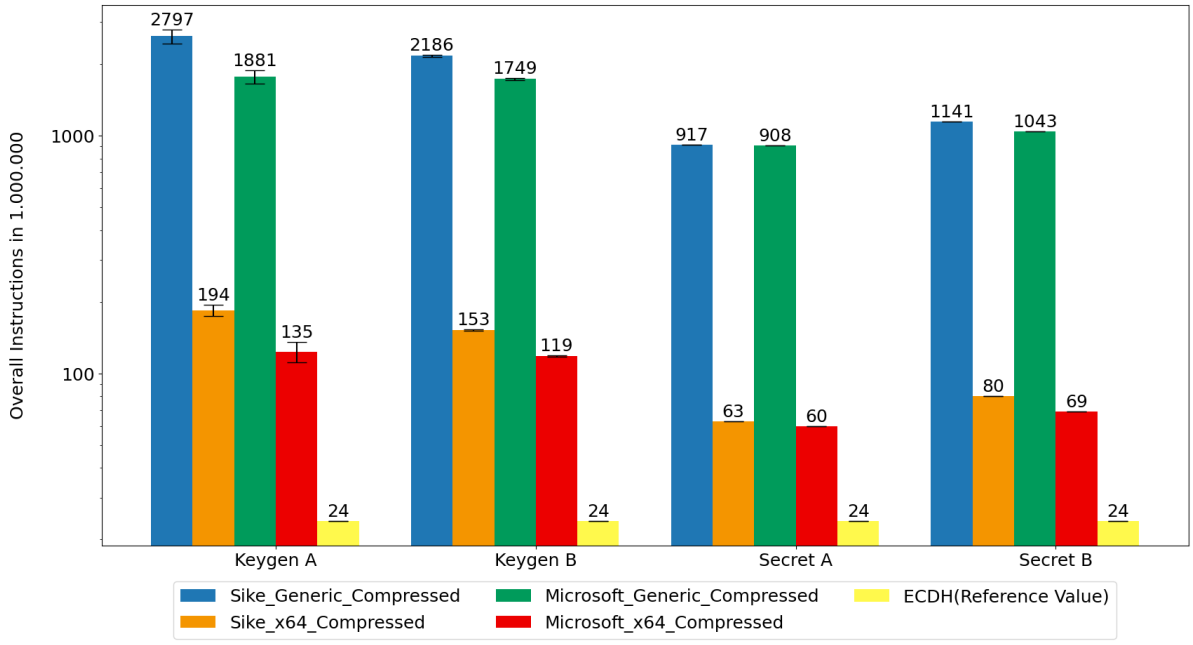

Figure 5.11.: Overall instructions for compressed implementations of SIKE and PQCrypto-SIDH using `p751` compared to ECDH via `secp521r1`.

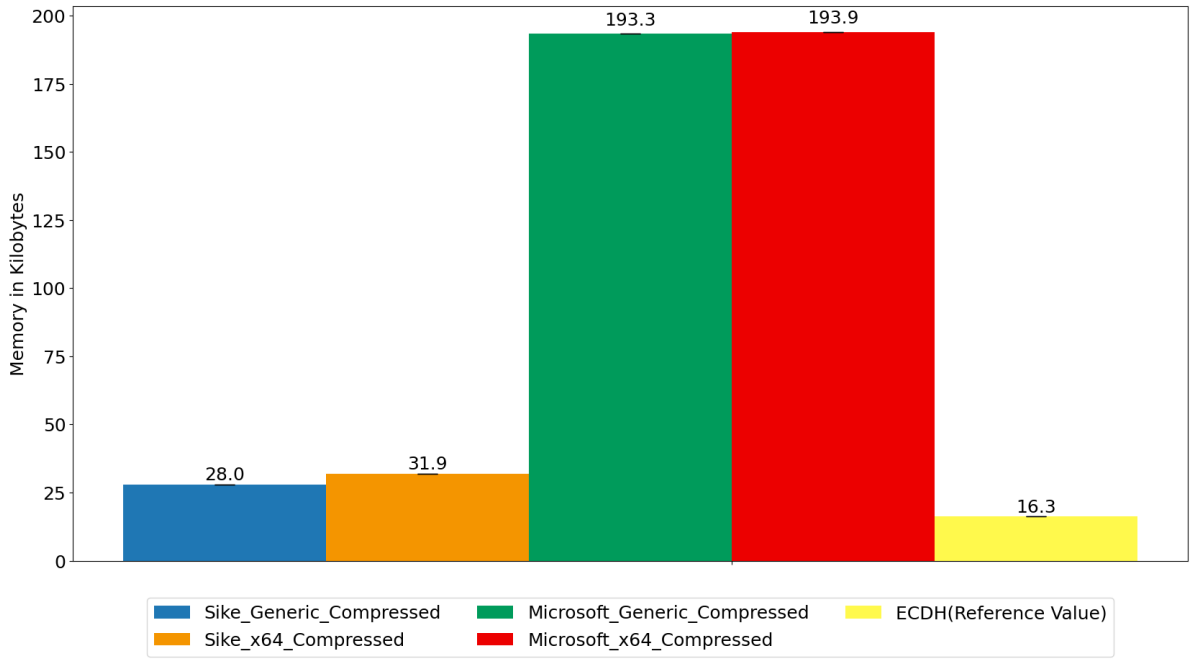

Figure 5.12.: Maximum memory consumption in kilobytes for compressed implementations of SIKE and PQCrypto-SIDH using `p751` compared to ECDH via `secp521r1`.

5.2.2. Summary of findings

The comparison among SIKE implementations reveals significant the performance differences: The x64 optimized version (`SIKE_x64`) is faster than the generic optimized variant (`SIKE_Generic`). The reference implementation (`SIKE_Reference`) is by far the slowest implementation. This is reasonable, since the reference implementation does not focus on performance. While `SIKE_Generic` implements code for arbitrary target architectures, `SIKE_x64` implements hardware specific instructions leading to faster execution. Moreover, compressed versions reduce the performance for SIDH key exchanges: In order to decrease cryptographic keys these implementations implement compression and decompression algorithms. These key transformations lead to an increased amount of executed instructions. Additionally, more memory is allocated during key (de)compression.
The evaluation between PQCrypto-SIDH and SIKE exemplarily shows, how further research might influence isogeny based cryptography. Improved compression algorithms reduced the executed instructions while the memory costs increased (space-time trade-off). At the same time, the analysis of optimized algorithms reveals the similarities between both libraries.

5.3. Comparing SIKE and CIRCL

This section compares the SIDH libraries SIKE and CIRCL. Since CIRCL does not provide any compressed versions, the analysis in this section includes the following implementations: `SIKE_Generic`, `SIKE_x64`, `CIRCL_Generic` and `CIRCL_x64`.
Firstly, the *generic optimized* implementations are compared in terms of executed instructions and allocated memory (subsection 5.3.1). Secondly, subsection 5.3.2 compares the *x64 optimized* variants of both libraries. Thirdly, subsection 5.3.3 evaluates the measured execution hotspots for SIKE and CIRCL in. Finally all findings in this section are summarized in subsection 5.3.4.

5.3.1. Generic optimized implementations

Generic optimized implementations do not make use of any assembler specific instructions. They implement the SIDH key exchange without any hardware assumptions resulting in algorithms that can be executed on multiple target architectures. This section compares `SIKE_Generic` and `CIRCL_Generic`.

Figure 5.13 contains the benchmarks for SIDH key exchanges using generic implementations instantiated with `p434`. In all four aspects of the key exchange `SIKE_Generic` processes twice as many instructions as `CIRCL_Generic`. This difference even grows when considering parameter set `p751` (Figure 5.15), where `SIKE_Generic` executes on average 2.3 times more instructions than `CIRCL_Generic`.
At the same time `CIRCL_Generic` allocates more memory: Running the benchmarks with parameter `p434` reveals a memory overhead of 16.2 kB (Figure 5.14). Using parameter `p751` reduces this overhead to 12.5 kB (Figure 5.16), however, this difference is still significant. Additionally, note the standard deviation for the allocated memory measured for `CIRCL_Generic` in both graphs: The generic optimized CIRCL implementation does not provide a constant memory consumption.

This analysis shows, that CIRCL provides a faster generic optimized implementation than SIKE. This result is a contrast to subsection 5.3.2, where the C implementation of SIKE is faster that the GO implementation of CIRCL. A detailed investigation reveals performance differences in time expensive hotspot functions. Both libraries spent most time for calculating *Karatsubas multiplication algorithm* (see subsection 5.3.3). The generic implementation of CIRCL calculates this algorithm on average in 1120 instructions, while the realization in generic C of SIKE executes on average 3373 instructions. Since these functions are called more than 130 000 times, these differences have strong impact to the overall executed instructions. Note that these values were extracted manually from the *callgrind* output files generated by the benchmarking suite.
At the same time `SIKE_Generic` occupies less memory – again a trade-off between executed instructions and allocated memory can be observed (e.g. [49]). The benchmarked deviation of peak memory consumption by CIRCL might be the result of the GO garbage collector [47].

5.3.2. X64 optimized implementations

X64 optimized implementations exploit specific hardware operations to improve performance for machines implementing the x64 instruction set. This section compares `SIKE_x64` and `CIRCL_x64`.

Figure 5.13 shows the execution time benchmarks for these optimized variants initiated with `p434`. In all four categories listed in the graph *SIKE_x64* is the fastest x64 implementation. *CIRCL_x64* is slower: For key generation CIRCL processes 9 million instructions more and for generating a shared secret the library generates 7 million additional instructions compared to SIKE. This corresponds to a relative difference of factor 1.5. Figure 5.15 compares the implementations initiated with *p751* - matching the highest NIST security level 5. *SIKE_x64* stays the faster version while the relative difference to *CIRCL_x64* (factor 1.36) decreases.
In order to compare the memory consumption of the x64 optimized implementations, consider Figure 5.14. The memory benchmarks of *SIKE_x64* (8.2 kB) are lower than the measured allocations of *CIRCL_x64*, revealing a peak allocation of 22.0 kB for a single SIDH key exchange. This is by factor 2.7 greater compared to SIKE. However, Figure 5.16 reveals that the memory consumption for *CIRCL_x64* does barely change for higher security classes. Nevertheless, *CIRCL_x64* has a more intense memory consumption than *SIKE_x64* using `p751` (by factor 1.7). Moreover, the measured peak memory consumption of CIRCL is not constant: Both graphs (5.14 and 5.16) show a considerable standard deviation during 100 runs of the x64 optimized algorithm.

The fastest x64 optimized SIDH key exchange is performed by *SIKE_x64*. Again, these results can be explained when considering the performance of hotspot functions: In contrast to the analysis of *generic optimized* versions in subsection 5.3.1, the *x64 optimized* implementation of SIKE is faster than CIRCL: SIKE's realization of *Karatsubas multiplication algorithm* (execution hotspot of both libraries) executes on average 280 instructions, while the CIRCL variant processes on average 357 instructions. This advantage results in reduced overall instructions for `SIKE_x64`.
CIRCL_x64 allocates clearly more memory and does not implement a constant amount of memory allocations. These dynamic memory allocations of the binary could be a result of the garbage collector used by the GO programming language [47].

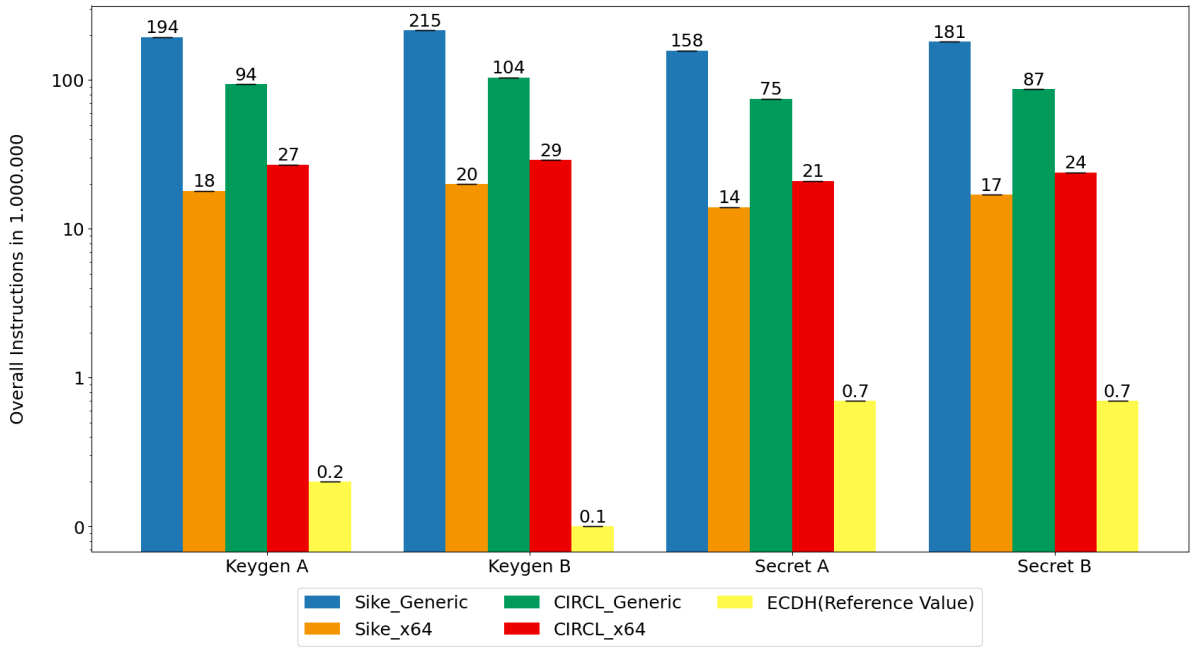

Figure 5.13.: Overall instructions for SIDH parameter `p434` compared to ECDH via `secp256r1`.

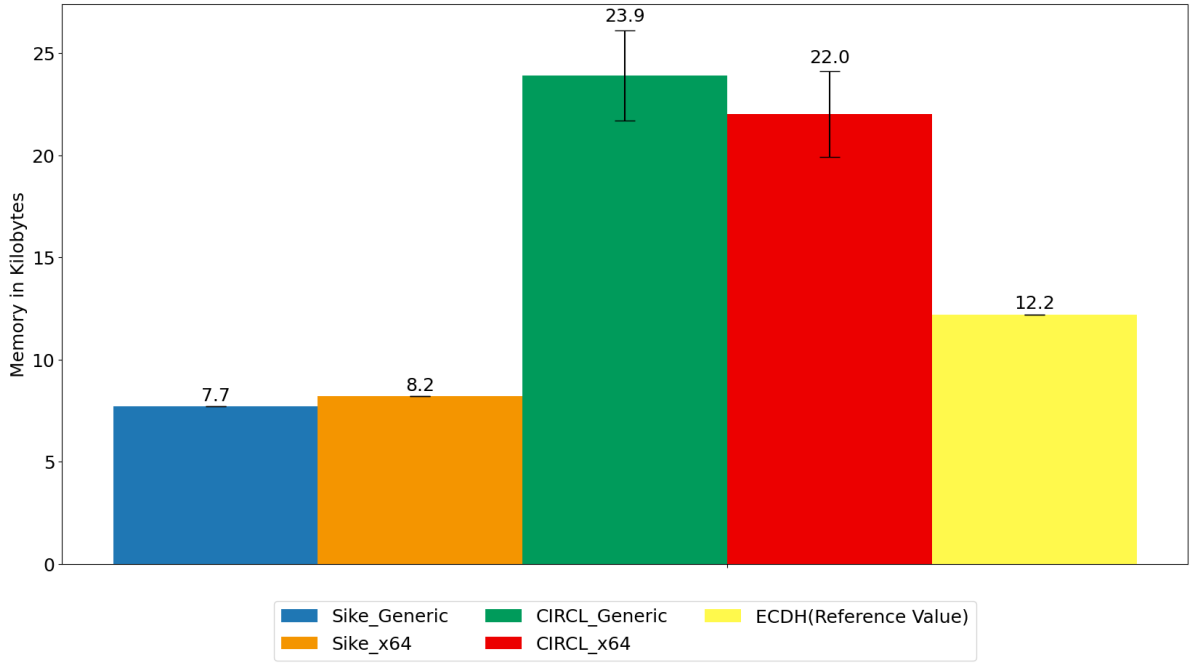

Figure 5.14.: Maximum memory consumption in kilobytes for SIDH parameter `p434` compared to ECDH via `secp256r1`.

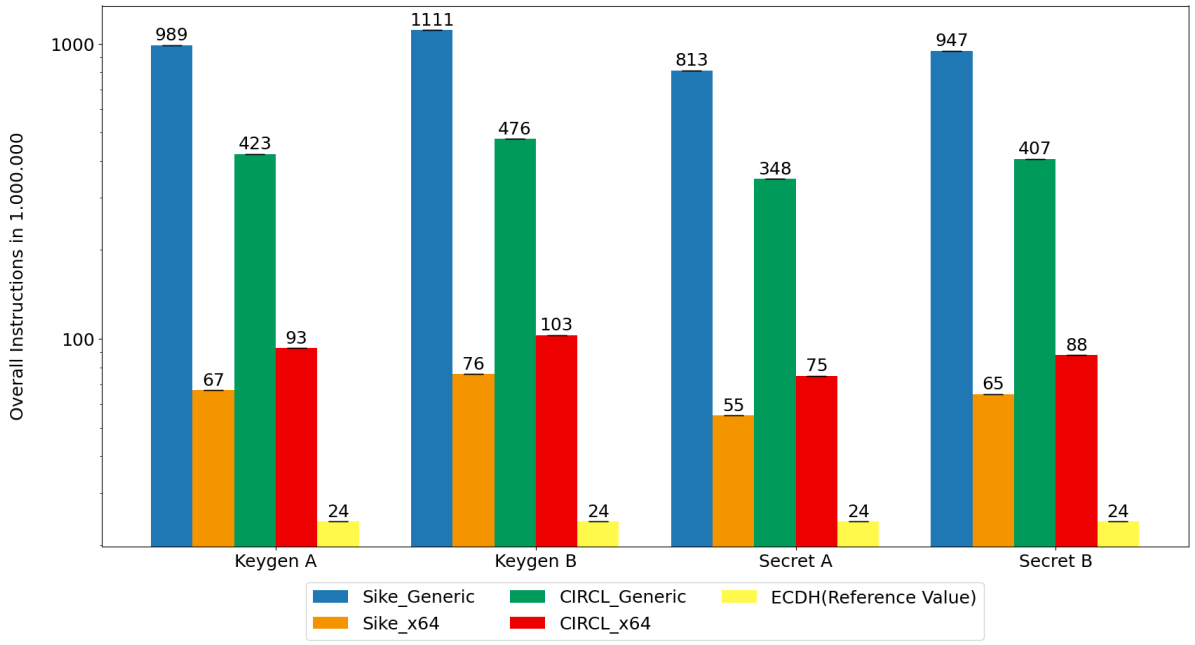

Figure 5.15.: Overall instructions for SIDH parameter `p751` compared to ECDH via `secp521r1`.

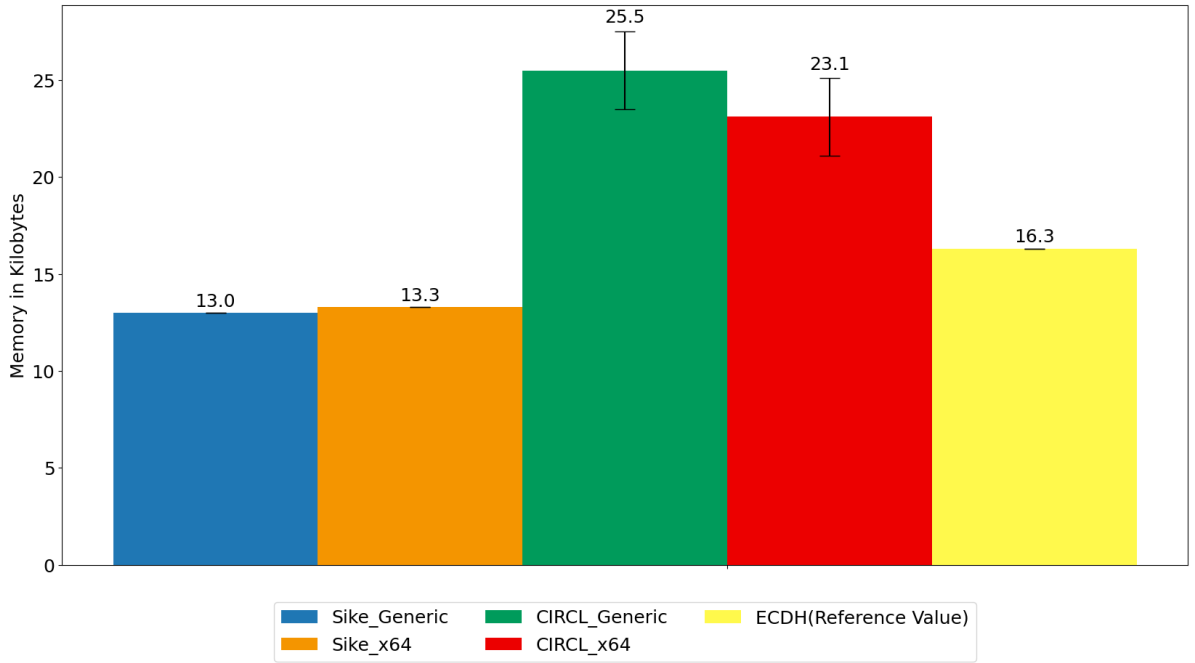

Figure 5.16.: Maximum memory consumption in kilobytes for SIDH parameter `p751` compared to ECDH via `secp521r1`.

5.3.3. Analysis of execution hotspots

Besides detailed benchmarks, addenda A.2 also lists the execution hotspots of each implementation initialized with different parameters. The following description of these hotspots reveals potentials to further improve performance of SIDH.
Each implementation of the SIKE library spends more than 50% of its execution time within the function `mp_mul`. The second great hotspot of the library 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` (~22%) where `xxx` $\in \{434, 503, 751\}$. The source code provides further information, however, the documentation of *CIRCL* makes it hard to form 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 $z = x * R^{-1} \pmod{2 * p}$ for given integers x and p (based on Montgomery reduction).

It can be seen that both 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), 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 (details in section 5.4).

5.3.4. Summary of findings

This section reveals that `CIRCL_Generic` is faster than `SIKE_Generic`. At the same time `SIKE_x64` executes less instructions than `CIRCL_x64` indicating `SIKE_x64` as the fastest measured SIDH key exchange in this thesis. The reason for different libraries being faster for different implementations lies in the performance of the highly relevant hotspot functions. While their realization for *generic optimized* implementations is more efficient in CIRCL, the *x64 optimized* algorithms of SIKE perform better.
In terms of allocated memory CIRCL claims more resources than SIKE for all implementations. As stated above, this is reasonable since GO implements a garbage collector for memory maintenance.

5.4. 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 ECDH. The following benchmark analysis is completed by an evaluation of ECDH hotspot functions (subsection 5.4.1). Finally this section summarizes all findings in subsection 5.4.2.

Besides all optimized SIDH versions, Figure 5.13 also shows benchmarks for a ECDH reference value via `secp256r1`. Note that the security class of parameter set `p434` matches with `secp256r1` (see section 4.2 for details). Compared to the fastest SIDH optimized implementation (`SIKE_x64`), ECDH is significantly faster: On average 90 times less instructions for KeygenA and 200 times less instruction for KeygenB are executed. Additionally, the generation of the secret is about 20 times faster for secretA and 24 times faster for secretB.
More moderate results can be observed for parameter set `P751` and `secp512r1` (Figure 5.15): KeygenA (2.8 times), KeygenB (3.1 times), SecretA (2.3 times) and SecretB (2.7 times) of `secp512r1` are faster compared to the fastest SIDH variant `SIKE_x64`.
While ECDH exploits much faster execution times, the memory consumption of ECDH is higher. Figure 5.14 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 5.16).

ECDH parameter sets enable faster computation times for a single key exchange. Especially the elliptic curve `secp256r1` executes significantly less instructions than the other implementations. This could be caused by the high interest of cryptographic research in the field of elliptic curves in the past years. As a result, ECDH provide highly optimized algorithms. Especially the *OpenSSL* implementation of `secp256r1` (providing a security level of 128 bit) benefits from these progresses [51]. A further analysis of the *callgrind* files generated by the benchmarking suite reveals that `secp256r1` is assembler optimized in *OpenSSL*, while `secp384r1` and `secp512r1` are not. This justifies the significant low number of executed instructions for `secp256r1`. At the same time ECDH suffers from an increased memory consumption compared to SIDH. The reason for this might be a trade-off between execution time and memory consumption for elliptic curve algorithms [51].

5.4.1. Analysis of ECDH execution hotspots

As listed in addenda A.2, the measured execution hotspots for the *OpenSSL* implementation of ECDH differ, since the *OpenSSL* implementation of `secp256r1` is highly optimized while `secp384r1` and `secp512r1` are not (see section 5.4 and [51] for details). Thus, the underlying source code of `secp256r1` is different leading to different observed execution hotspots.
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 accelerate cryptographic primitives. This results in fast and user-friendly encryption schemes.

5.4.2. Summary of findings

The analysis in this section showed that modern ECDH algorithms execute significantly less instructions than SIDH for a single key exchange, while allocating more memory (space-time trade-off). Hence, the deployment of SIDH in a wide range of applications is currently hard to imagine. At the same time research is still ongoing and multiple optimizations for SIDH were proposed within the last years (see `https://sike.org/`).
All optimized implementations (SIDH and ECDH) are based on the same highly optimized prime filed arithmetic algorithms. Thus, isogeny based cryptography might need to reduce arithmetic operations in general, since major optimizations of the already well-researched prime filed arithmetic are rather unlikely.

5.5. 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 5.5.1. The implemented key sizes of all SIDH libraries and the used ECDH curves will be considered additionally in subsection 5.5.2.

5.5.1. Constant time

Besides the average for $N = 100$ executions addenda A.2 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 which 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 in addenda A.2.

The standard deviation for the following variants is zero and thus these variants implement constant time cryptography for all parameter sets:

- `SIKE_Reference`
- `SIKE_Generic`
- `SIKE_x64`
- `ECDH` initialized with curve `secp256r1`.
  Note that the standard deviation for `secretA` and `secretB` is not zero. However, when analyzing the detailed results of the benchmarking suite the following can be observed: 99 runs of the function `secretA` execute *exactly* 652 796 instructions. Only one execution of this function executed 652 777 instructions (19 instructions less). The same behavior can be observed for `secretB`: Exactly one run of this function executes 19 instructions less than the other 99 runs. Since this difference corresponds to about 6 clock cycles on a 3GHz processor and only one run out of 99 is affected, one can state that this is a constant time implementation. Both functions `secretA` and `secretB` make use of the same *OpenSSL* function `ECDH_compute_key` (each passing appropriate parameters).

On the other hand, the following implementations show deviations in their execution times. Thus, they are not implemented in constant time:

- `ECDH` for the benchmarked curves `secp384r1` and `secp512r1` in *OpenSSL*.
  Note that the source code of the benchmarking suite contains the C-file performing the ECDH key exchange (`container/ECDH/benchmark.c`). The APIs used in this file are adapted from the official *OpenSSL* wiki[1]. The concrete functions of that API that do not execute a static amount of instructions for the parameters above are namely: `EC_KEY_generate_key` and `ECDH_compute_key`.
- `CIRCL_Generic` and `CIRCL_x64`
  In particular the API functions `GeneratePublicKey` and `DeriveSecret` (both are class functions for private key objects, for details see the CIRCL API in section 3.2) do not execute a static amount of instructions.
- `SIKE_Generic_Compressed` and `SIKE_x64_Compressed`
  This is expected, since compressed versions need to compress and decompress the public key. This compression depends on the public key, which depends on a randomly chosen private key.

[1] Section *"Using the Low Level APIs"* at `https://wiki.openssl.org/index.php/Elliptic_Curve_Diffie_Hellman`

5.5.2. Key size

This section compares the size of public keys implemented by the SIDH libraries *SIKE* and *CIRCL* with modern *OpenSSL* ECDH. The used parameters matching the appropriate NIST security level can be found in section 4.2. 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 size of the used ECDH curves is part of their 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 5.2.: 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 [35].

6. Conclusion

This final chapter presents the major results of the previous chapter 5 in section 6.1. The considerations in section 6.2 describe the limitations of this thesis.

6.1. Results

This section summarizes all results of the benchmarking suite. For a detailed analysis of the measured data see chapter 5 and addenda A.2. The presented data in this section provides a relative comparison between the SIDH libraries *SIKE* and *CIRCL*. Moreover, the relative difference between SIDH and ECDH is shown.
The most efficient implementation in the following tables is highlighted in green and has a relative difference of 1 to itself. All other implementations in the appropriate row are labeled with the relative distance to the most efficient implementation. For this evaluation the execution times and memory consumption for a whole Diffie-Hellman exchange are accumulated (key generation of A and B *plus* secret generation of A and B).

6.1.1. Comparing SIDH libraries

This section summarizes the comparison between SIKE and CIRCL. The tables below list the measured results considering all available parameter sets. *Generic optimized* and *x64 optimized* implementations are listed separately.

Generic optimized

The values in Table 6.1 indicate CIRCL to be the fastest *generic optimized* implementation. For all parameters included in CIRCL the library executes less instructions than SIKE (`p610` is not available in CIRCL). At the same time the *compressed* version of SIKE processes the most operations of all *generic optimized* variants.
Table 6.2 lists the benchmarking results considering memory consumption for *generic optimized* implementations. SIKE is the most efficient variant allocating less memory than the fastest version CIRCL.

Generic optimized			
Parameter \ Library	SIKE	SIKE compressed	CIRCL
p434	2.07	3.76	1
p503	3.2	5.75	1
p610	1	1.79	-
p751	2.33	4.13	1

Table 6.1.: Relative comparison of executed instructions regarding *generic optimized* SIDH implementations.

Generic optimized			
Parameter \ Library	SIKE	SIKE compressed	CIRCL
p434	1	2.2	3.1
p503	1	2.45	3.12
p610	1	2.12	-
p751	1	2.15	1.96

Table 6.2.: Relative comparison of memory consumption regarding *generic optimized* SIDH implementations.

X64 optimized

The summary for *x64 optimized* SIDH implementations is listed in Table 6.3. SIKE executes the least amount of instructions for all parameter sets. CIRCL is slightly slower and the *compressed* version of SIKE executes the most operations. Table 6.3 additionally indicates SIKE to claim at least memory for a single SIDH key exchange. CIRCL and SIKE *compressed* allocate an equal amount of memory for lower security levels. However, for increasing security levels the distance between CIRCL and SIKE decreases.

x64 optimized			
Parameter \ Library	SIKE	SIKE compressed	CIRCL
p434	1	1.87	1.45
p503	1	2.57	1.65
p610	1	1.75	-
p751	1	1.81	1.35

Table 6.3.: Relative comparison of executed instructions regarding *x64 optimized* SIDH implementations.

x64 optimized			
Parameter \ Library	SIKE	SIKE compressed	CIRCL
p434	1	2.36	2.69
p503	1	2.54	2.54
p610	1	2.39	-
p751	1	2.4	1.73

Table 6.4.: Relative comparison of memory consumption regarding *x64 optimized* SIDH implementations.

6.1.2. Comparing SIDH and ECDH

The analysis of Table 6.5 reveals the differences between *SIDH* and *ECDH* in terms of execution times. Note that all security classes and only the x64 optimized variants of all libraries are considered. While the overhead of *SIDH* for the NIST security level 1 is enormous, the relative difference for higher security levels decreases.
While ECDH is faster in terms of executed instruction, the library also requests more memory than SIDH (Table 6.6).

NIST Security	SIDH				ECHD	
	SIDH parameter	SIKE	SIKE compressed	CIRCL	ECDH parameter	openSSL
Level 1	p434	44.55	81.61	64.46	secp256r1	1
Level 3	p610	4.18	7.48	-	secp256r1	1
Level 4	p751	2.71	4.91	3.66	secp256r1	1

Table 6.5.: Relative comparison of executed instructions between ECDH and SIDH.

NIST Security	SIDH				ECHD	
	SIDH parameter	SIKE	SIKE compressed	CIRCL	ECDH parameter	openSSL
Level 1	p434	1	2.36	2.69	secp256r1	1.49
Level 3	p610	1	2.39	-	secp256r1	1.23
Level 4	p751	1	2.4	1.73	secp256r1	1.22

Table 6.6.: Relative comparison of allocated memory between ECDH and SIDH.

6.2. Limitations of this thesis

This section lists limitations and approaches for future work in the context of this thesis. Firstly, the correctness of the results measured by the benchmarking suite are based on the tools *massif* and *callgrind* provided by `valgrind`. Any disruptions within these tools directly

affect all benchmarks.
Secondly, the integration process to generate benchmarks for a new implementation could be further improved. Up to now, the process described in 4.2.3 is time-consuming and error-prone. This could be improved e.g. by auto detecting new sub folders containing appropriate Makefiles. Moreover, the generation of different output graphs can currently only be achieved by manually changing the source code of the benchmarking suite. Appropriate command line arguments for the benchmarking suite might increase usability.
Finally, for the comparison between SIDH and ECDH only three curves of the *OpenSSL* library are considered. A more precise analysis among existing ECDH libraries might reveal more detailed results.

A. General Addenda

A.1. Project hierarchy

This section illustrates the project hierarchy of the developed benchmarking suite. For details see the description below.

```
root
├── container
│   ├── SIKE
│   ├── Microsoft
│   ├── CIRCL
│   ├── ECDH
│   ├── helper
│   ├── src
│   │   ├── test
│   │   ├── utils
│   │   │   ├── benchmarks.py
│   │   │   ├── caching.py
│   │   │   ├── callgrind.py
│   │   │   ├── plot_graph.py
│   │   │   └── plot_tables.py
│   │   ├── base.py
│   │   ├── circl.py
│   │   ├── ecdh.py
│   │   ├── microsoft.py
│   │   └── sike.py
│   ├── requirements.txt
│   └── benchmarking.py
├── data
│   ├── XXX.png
│   ├── XXX_mem.png
│   ├── cached.json
│   ├── result.html
│   └── result.tex
├── Dockerfile
├── README.md
└── run.sh
```

The root project directory contains following sub directories:

1. `container`:
   This folder contains everything needed to start the Docker container. Besides the source code of the benchmarking suite (`container/src`), this folder also contains sub folders for all included libraries. These sub folders (`container/SIKE`, `container/Microsoft`, `container/CIRCL` and `container/ECDH`) contain Makefiles to compile and benchmark the appropriate library (for details see subsection 4.2.3).
   Requirements.txt lists the Python dependencies for the benchmarking suite and *benchmarking.py* contains the entrypoint of the benchmarking suite.

2. `data`:
   The data folder contains the output data of the benchmarking suite as described in section 4.3.

3. `Dockerfile`:
   This Dockerfile specifies the Docker container the benchmarking suite runs in.

4. `README.md`:
   The README describing the usage of the benchmarking suite.

5. `run.sh`:
   This shell script can be invoked with the following arguments: `benchark`, `build` and `test`. For details see section 4.3.

A.2. Detailed Benchmarks

This addenda contains the detailed benchmarks which were measured by the benchmarking suite. The listed tables contain benchmarking results for all supported 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.2.1. 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.2.2. 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.2.3. 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.2.4. 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.2.5. 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.2.6. 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.2.7. 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.7.: 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.2.8. 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.8.: 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.2.9. 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.9.: 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.2.10. 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.10.: 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%

A.2.11. Benchmarks for CIRCL x64

Parameter	434	503	751
PrivateKeyA	2.823 (716)	2.897 (794)	2.915 (779)
PublicKeyA	27.586.964 (8.609)	31.016.984 (10.675)	93.010.442 (9.883)
PrivateKeyB	663 (0)	663 (0)	663 (0)
PublicKeyB	29.068.891 (463)	32.900.437 (520)	103.305.745 (699)
SecretA	21.166.569 (1.784)	24.018.082 (998)	75.274.951 (541)
SecretB	24.454.234 (339)	27.733.879 (407)	88.023.885 (355)
Memory in bytes	21.992 (2.120)	21.620 (1.810)	23.116 (2.006)

Table A.11.: Benchmarks for CIRCL x64

Execution hotspots parameter *434*:

1. `p434.mulP434`: 46.96%
2. `p434.rdcP434`: 22.78%
3. `p434.subP434`: 5.77%

Execution hotspots parameter *503*:

1. `p503.mulP503`: 41.12%
2. `p503.rdcP503`: 22.89%
3. `p503.subP503`: 8.34%

Execution hotspots parameter *751*:

1. `p751.mulP751`: 56.01%
2. `p751.rdcP751`: 21.66%
3. `p751.subP751`: 6.01%

A.2.12. Benchmarks for CIRCL Generic

Parameter	434	503	751
PrivateKeyA	2.749 (678)	2.754 (692)	2.806 (704)
PublicKeyA	94.855.793 (9.351)	139.394.045 (8.178)	423.781.686 (10.860)
PrivateKeyB	663 (0)	663 (0)	663 (0)
PublicKeyB	104.075.408 (761)	152.890.633 (837)	476.666.916 (773)
SecretA	75.854.959 (1.654)	112.603.650 (1.000)	348.366.025 (782)
SecretB	87.814.021 (437)	129.337.846 (501)	407.146.301 (640)
Memory in bytes	23.903 (2.246)	24.312 (2.269)	25.503 (1.987)

Table A.12.: Benchmarks for CIRCL Generic

Execution hotspots parameter *434*:

1. `p434.mulP434`: 42.86%
2. `p434.rdcP434`: 33.77%
3. `p434.subP434`: 7.01%

Execution hotspots parameter *503*:

1. `p503.mulP503`: 43.8%
2. `p503.rdcP503`: 35.49%
3. `p503.subP503`: 6.35%

Execution hotspots parameter *751*:

1. `p751.mulP751`: 47.46%
2. `p751.rdcP751`: 37.86%
3. `p751.subP751`: 4.77%

List of Figures

List of Tables

List of Abbreviations

Notation	Description
AES	Advanced Encryption Standard, symmetric encryption scheme
CIRCL	Cloudflare Interoperable, Reusable Cryptographic Library
CPU	Central Processing Unit, the processor of a computer
Dec	Decryption function
DH	Diffie-Hellman key exchange protocol
DSA	Digital Signature Algorithm
ECDH	Elliptic Curve Diffie-Hellman key exchange protocol
ECDSA	Elliptic Curve Digital Signature Algorithm
Enc	Encryption function
HTML	Hypertext Markup Language
IPC	Instrucitons per Cycle, the executed instructions within a CPU clycle
ISA	Instruction Set Arcitecture
JVM	Java Virtual Maschine
k_{AB}	Shared secret between subjects A and B
kB	kilobytes
KEM	Key Encapsulation Mechanism
McEliece	Public-key cryptosystem
MSS	Merkle Signature Scheme
NIST	National Institute of Standards and Technology
NTRU	Public-key cryptosystem, NTRUencrypt

Notation	Description
NTRUsign	Digital signature algorithm
OpenSSL	Software library for secure communication
PFS	Perfect Forward Secrecy
PKE	Public-Key Encryption
PQCrypto-SIDH	Post-Quantum cryptographic library based on Supersingular Isogenies
RSA	Rivest–Shamir–Adleman public-key cryptosystem
RSA-KEM	Key Encapsulation Mechanism based on RSA
SHA	Secure Hash Algorithms
SIDH	Supersingular Isogeny Diffie-Hellman key exchange protocol
SIKE	Supersingular Isogeny Key Encapsulation, first proposed cryptographic protocols based on isogenies including key exchange, public-key encryption and key encapsulation

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] D. R. L. Brown. *Breaking RSA May Be As Difficult As Factoring*. Journal of Cryptology, Report 2005/380. https://eprint.iacr.org/2005/380. 2005.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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).

[14] E. Barker and A. Roginsky. *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. NIST, National Institute of Standards and Technology, 2019.

[15] NIST. *Post-Quantum Cryptography - Call for Proposals*. 2017 (accessed November 14, 2020). URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/Call-for-Proposals.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

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

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

[26] 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.

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

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

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

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

[31] D. Hofheinz, K. Hövelmanns, and E. Kiltz. *A Modular Analysis of the Fujisaki-Okamoto Transformation*. Cryptology ePrint Archive, Report 2017/604. https://eprint.iacr.org/2017/604. 2017.

[32] 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.

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

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

[35] 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.

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

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

[38] Wultra. *Sike for Java*. `https://github.com/wultra/sike-java`. 2020.

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

[40] *GoLang Wiki Compiler Optimizations*. 2020 (accessed October 25, 2020). URL: `https://github.com/golang/go/wiki/CompilerOptimizations`.

[41] A. R. Alameldeen and D. A. Wood. "IPC considered harmful for multiprocessor workloads". In: *IEEE Micro* 26.4 (2006), pp. 8–17.

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

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

[44] A. K. Lenstra. *Key lengths*. Tech. rep. Wiley, 2006.

[45] G. Gupta. "What is Birthday attack??" In: (2015).

[46] E. Barker. *Recommendation for Key Management, Part 1: General*. Tech. rep. National Institute of Standards and Technology, 2016.

[47] R. Hudson. *Getting to Go: The Journey of Go's Garbage Collector*. 2018 (accessed November 26, 2020). URL: `https://blog.golang.org/ismmkeynote`.

[48] GO. *GO FAQ - Garbage Collection*. 2020 (accessed November 28, 2020). URL: `https://golang.org/doc/faq#garbage_collection`.

[49] M. Hellman. "A cryptanalytic time-memory trade-off". In: *IEEE Transactions on Information Theory* 26.4 (1980), pp. 401–406. DOI: `10.1109/TIT.1980.1056220`.

[50] G. C. C. F. Pereira, J. Doliskani, and D. Jao. *x-only point addition formula and faster torsion basis generation in compressed SIKE*. Cryptology ePrint Archive, Report 2020/431. `https://eprint.iacr.org/2020/431`. 2020.

[51] M. Adalier et al. “Efficient and secure elliptic curve cryptography implementation of Curve P-256”. In: *Workshop on Elliptic Curve Cryptography Standards*. Vol. 66. 2015.