DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking Supersingular Isogeny Diffie-Hellman Implementations

Jonas Hagg

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking Supersingular Isogeny Diffie-Hellman Implementations

Benchmarking Supersingular Isogeny Diffie-Hellman Implementierungen

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

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

Munich, Submission date Jonas Hagg

Acknowledgments

Abstract

Kurzfassung

Contents

1 Background

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

Symmetric Cryptography

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

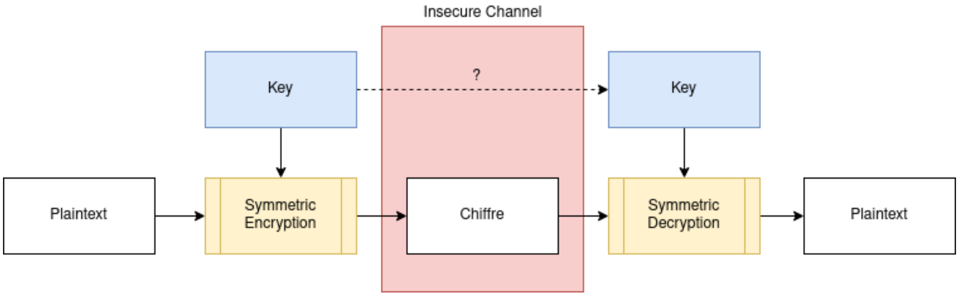

Figure 1.1: Simple symmetric encryption scheme: Encryption and decryption algorithm use the same key.

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

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

Asymmetric Cryptography

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

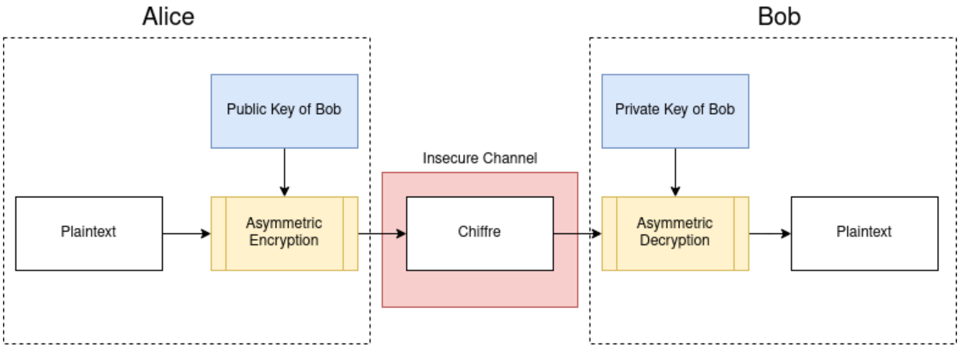

Figure 1.2: Asymmetric encryption scheme: Encryption and decryption algorithm use different keys.

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

$$\begin{aligned} Enc(p, e_{Bob}) &= c \\ Dec(c, d_{Bob}) &= p \end{aligned}$$

In contrast to the symmetric encryption, no secret key needs to be exchanged. However, the encryption and decryption of data using asymmetric encryption require intensive mathematical computations. Hence, the encryption of big sets of data using asymmetric encryption is not efficient. One the other hand, symmetric encryption algorithms are usually based on simple operations, such as bit shifting or XOR. This can be implemented efficiently in software and hardware. Thus, the practical relevance of symmetric encryption is enormous [**ITSicherheit**]. As stated above, securely exchanged keys are a precondition for the use of efficient symmetric encryption schemes. In order to exchange arbitrary keys securely, different key exchange protocols are available.

1.1 Key Exchange

1.1.1 Diffie-Hellman Key-Exchange

The Diffie-Hellman key exchange was introduced by Whitfield Diffie and Martin Hellman in 1976 [**diffie1976new**]. The resulting shared key of the protocol is calculated decentralized and is never transported over an insecure channel.

Protocol

The classical Diffie-Hellman key exchange assumes, that Alice and Bobs want to create a shared secret key. Therefore, they agree on a big prime p and g, which is a primitive root modulo p[1]. Both, p and g are not secret and may be known to the public [**watjen2018kryptographie**].

1. Alice choses a random $a \in \{1, 2, ..., p-2\}$ as private key.
2. Alice calculates the public key $A = g^a modp$.
3. Bob choses a random $b \in \{1, 2, ..., p-2\}$ as private key.
4. Bob calculates the public key $B = g^b modp$.
5. Alice and Bob exchange their public keys A and B.
6. Alice calculates:
$$\begin{aligned} k_{AB} &= B^a modp \\ &= (g^b modp)^a modp \\ &= g^{ab} modp \end{aligned} \tag{1.1}$$
7. Bob calculates:
$$\begin{aligned} k_{AB} &= A^b modp \\ &= (g^a modp)^b modp \\ &= g^{ba} modp \\ &= g^{ab} modp \end{aligned} \tag{1.2}$$
8. Alice and Bob created the shared secret k_{AB}. Note that only the public keys of Alice and Bob were send over an insecure channel. The generated secret was calculated decentralized by Alice and Bob.

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

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

$$\begin{tikzcd}
g \arrow[r, "x \longrightarrow x^a"] \arrow[d, "x \longrightarrow x^b"'] & g^a \arrow[d, "x \longrightarrow x^b"] \\
g^b \arrow[r, "x \longrightarrow x^a"'] & g^{ba} = g^{ab}
\end{tikzcd}$$

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

Security

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

$$\begin{aligned} b &= log_g\, B \; mod \; p \\ a &= log_g\, A \; mod \; p \end{aligned}$$

Hence, the security of the classical Diffie-Hellman key exchange is based on the discrete logarithm problem (see **??**). When using ephemeral key pairs, the Diffie-Hellman key exchange may be used as efficient perfect forward secrecy (PFS) protocol [**ITSicherheit**].

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 and point P on the elliptic curve. The computations to generate the shared secret k_{AB} follow the same principles, but obviously they are adopted to work on elliptic curves. The advantage of ECDH is the increased security strength while using the same key length than the classical Diffie-Hellman protocol [**ITSicherheit**].

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

1.1.2 Key Encapsulation

Key encapsulation methods (KEM) transmit a previously generated symmetric key. KEMs usually use asymmetric key pairs in order to encrypt the generated symmetric key. In the following the concept is shown by the RSA based PKCS #1 v1.5 algorithm, which uses RSA key pairs to transmit a shared secret from Alice to Bob [**rsakem**]:

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

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

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

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

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

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

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

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

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

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

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

1.1.3 Differences

TODO: differences betweent kem and kex (decentralized, pfs?), discrete log vs factorization

1.2 Post-Quantum Cryptography

In the following a *classical computer* refers to a non-quantum computer, which can be simulated by a deterministic Turing machine. In contrast to *classical computer* the term *quantum computer* describes a machine, which uses quantum mechanical phenomena to perform computations. It is important to note, that quantum computers can simulate classical computers. In addition, classical computer are able to simulate quantum computers with exponential time overhead. Thus, classical and quantum computations can calculate the same class of functions. However, quantum computers enable operations, that allow much faster computation [**nielsen2002quantum**].
In the past, scientists queried, if large-scale quantum computer are a physical possibility. It was stated, that the underlying quantum states are too fragile and hard to control. [**chen2016report**] Today, quantum error correction codes are known, which put a large-scale quantum computers within the realms of possibility [**lidar2013quantum**]. However, it is still a big engineering challenge from a laboratory approach to a general-purpose quantum computer, which involves thousands or millions of physical qubits [**chen2016report**].

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) [**chen2016report**]. These problems are theoretically solvable, but the computation of a result on classical computer claims an impractical amount of resources. In 2019, scientists solved the factorization problem for a 240 digit integer in about 900 core-years on a classical computer (one core year means running a CPU for a full year) [**boudot2795**]. In the following the discrete logarithm problem and the factorization problem are described.

Discrete Logarithm Problem

The discrete logarithm problem is the following challenge [**beutelspacher2010diskrete**]: Given a prim p and two integers g and y. Find an integer x, such that

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

Until today, it is not known if a classical computer is able to compute the general discrete logarithm problem in polynomial time. Thus, the discrete logarithm problem is considered to be difficult so solve for classical computers [**beutelspacher2010diskrete**]. This assumption makes the discrete logarithm problem an attractive basis for DSA, ElGamal, the classical Diffie-Hellman protocol, and for the elliptic curve Diffie-Hellman (ECDH).

Factorization Problem

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

$$n = p \star q$$

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

1.2.1 Impact of Quantum Computers on Cryptography

As stated above, quantum computers enable new operations which speed up certain algorithms. Two quantum algorithms which have enormous consequences on modern cryptography are *Shor's algorithm* and *Grover's algorithm* [**nielsen2002quantum**].

Shor's Algorithm

Peter Shor published *"Algorithms for quantum computation: discrete logarithms and factoring"* in 1994 [**shor1994algorithms**], where he demonstrated that the factorization problem and the discrete logarithm problem can be solved in polynomial time on quantum computers. Both problems are the basis of many public-key systems (RSA, DH, ECDH, ...), which are used intensively in modern communication systems. Hence, a quantum computer running *Shor's algorithm* would qualify the assumption of most asymmetric encryption schemes and thus break their security.

Grover's Algorithm

The second algorithm having impact on computer security was published by Lov Grover in 1996 (*"A fast quantum mechanical algorithm for database search"*, [**grover1996fast**]) - namely *Grover's algorithm*. The algorithms solves the problem of finding an element y in a set s (e.g. a database) where $|s| = N$. On a classical computer an algorithm solving the problem runs in $\mathcal{O}(N)$, however *Grover's algorithm* has complexity $\mathcal{O}(\sqrt{N})$ [**nielsen2002quantum**].
In contrast to public-key systems, which relay on hard mathematical problems, symmetric encryption schemes relay on the secrecy of a randomly generated key. Thus, to break symmetric encryption one need to perform a brute-force attack on the symmetric key. Using *Grover's algorithm* offers a square root speed up on classical brute force attacks [**mavroeidis2018impact**]. Assume a randomly generated n-bit key. A classical brute force algorithm lies in $\mathcal{O}(2^n)$, which is considered to be safe for a big n (e.g. n=128). *Grover's algorithm* speeds up this attack to $\mathcal{O}(\sqrt{2^n})$ = $\mathcal{O}(2^{n/2})$ [**mavroeidis2018impact**]. However, the complexity is still exponential and with a growing key size n the security can be further increased. Thus, *Grover's algorithm* forces symmetric encryption schemes to increase their used key size in order to stay secure.

To sum up, quantum computers make use of quantum mechanical phenomena in order to solve mathematical problems, which are assumed to be difficult for modern computers. As a result large-scale quantum computers might break many algorithms of modern asymmetric cryptography and enforce increased key sizes for symmetric encryption schemes. The following table form the NIST *"Report on Post-Quantum Cryptography"* [**chen2016report**] shows the impact of quantum computers on modern encryption schemes:

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

Table 1.1: Impact of quantum computers on modern encryption schemes

In contrast to this development in modern cryptography NIST states [**chen2016report**]:

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

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

1.2.2 Classes of Post-Quantum Cryptography

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

Lattice-based Cryptography

Lattice-based cryptography is - as the name suggests - based on the mathematical construct of lattices[2]. There are different computational optimization problems involving lattices, which are considered to be hard to solve even for quantum computers [**chi2015lattice**]. In 1998, NTRU was published as the first public-key system based on lattices [**hoffstein1998ntru**]. 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 [**gentry2009fully**].
Lattice-based cryptography is characterized by simplicity and efficiency [**chen2016report**]. The security of existing implementations (NTRU or Ring-LWE) can be reduced to NP-hard problems. However, lattices encryption schemes have problems to prove security against known cryptoanalysis [**chen2016report**].

Multivariate Cryptography

Multivariate cryptography are public key systems that are based on multivariate polynomials (e.g. $p(x,y) = x + 2y$) over a finite field $\mathbb{F}$. Their security is based on the prove,

[2] A lattice l is a subgroup of $\mathbb{R}^n$. In the context of cryptography usually integer lattices are considered: $l \subseteq \mathbb{Z}^n$ [**chi2015lattice**].

that solving systems of multivariate polynomials are NP-hard [**hartmanis1982computers**]. 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 [**ding2017current**], e.g. the Rainbow signature scheme [**ding2005rainbow**].

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 there errors and to reconstruct the plaintext [**bernstein2017post**]. McEliece, published in 1978, was the first of those systems and it has not been broken until today [**mceliece1978public**]. Beside asymmetric cryptography, code-based schemes have been proposed for digital signatures, random number generators and cryptographic hash functions [**bernstein2017post**].

Hash-based Signatures

Hash-based signatures describe the construction of digital signatures schemes based on hash functions. Thus, the security of theses primitives is based on the security of the underlying hash function and not on hard algorithmic problems [**bernstein2017post**]. Since hash functions are widely deployed in modern computer systems, the security of hash-based signatures is well understood [**chen2016report**].
The initially developed One-Time Signatures have the downside, that a new public key pair is needed for each signature [**becker2008merkle**]. In 1979, Merkle introduced the Merkle Signature Scheme (MSS), which uses one public key for multiple signatures [**merkle1979secrecy**]. Further improvements of MSS introduced public keys, which can be used for 2^{80} signatures. However, this also leads to longer signature sizes [**becker2008merkle**].

1.2.3 Isogeny-based Cryptography

Isogeny-based cryptography was proposed in 2011 as a new cryptographic system, that might resists quantum computing [**jao2011towards**]. Beside the publication describing isogeny-based cryptgraphy, the authors also provided a reference implementaion of a public key system, which is called SIKE [**sike2020spec**]. Isogeny-based cryptography benfits from small key sizes compared to other post-quantum cryptography classes, however, their performance is comparatively slow [**sike2020spec**]. The security of these primitives is based on finding isogenies between supersingular elliptic curves.
In the following, the problem is roughly illustrated. It is not indented to precise the mathematics behind isogeny-based cryptography, since this is not the scope of this work. However, the reader might become a little understanding of the magic behind supersingular isogenies. Next, the central component of isogeny-based cryptography, namely the Supersingular Isogeny Diffie Hellman (SIDH), is described. Finally, details of the reference implementation SIKE will be given.

Illustration of the Problem [urbanik2017friendly] [costello2019supersingular]

Isogeny-based cryptography works on supersingular elliptic curves. To be more precise, it is based on isogenies between supersingular elliptic curves.
In the context of elliptic curves, one can calculate a quotient of an elliptic curve E by a subgroup S. This essentially means to construct a new elliptic curve $E \backslash S$. Beside this new curve, the procedure also yields a function $\phi_S : E \rightarrow E \backslash S$, which is called *isogeny*. Carefully chosen elliptic curves E have a wide range of subgroups, which can be used to construct many isogenies.
Supersingular elliptic curves are a special type of elliptic curves, which have 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 certain curves, as described above.
The idea behind isogeny-based cryptography might be illustrated as followed:

1. Start with a known curve E and build a isogeny to a arbitrary reachable curve E_A.
2. This yields the isogeny $\phi_A : E \rightarrow E_A$, which is used as private key.
3. The curve E_A is used as part of the public key.

As usually in asymmetric cryptography, the hard mathematical problem is the calculation of the private key while knowing the public key. To be more precise: Find the isogeny $\phi_A : E \rightarrow E_A$ while knowing curves E and E_A is considered to be a quantum-resistant challenge. In literature, this is formally defined as *SIDH problem* [**sike2020spec**].
This key pair, however, is not used to decrypt or encrypt data. In fact, the procedure is very similar to the previously introduced Diffie-Hellman key exchange, where the key pair is used to establish a shared secret between two communication partners. In its core, isogney-based cryptography creates a shared secret via a Diffie-Hellman like procedure. This is called *Supersingular Isogeny Diffe Hellman (SIDH).*

Supersingular Isogeny Diffie Hellman (SIDH)

Recall the Diffie Hellman diagram in **??**, where the commutative property of the protocol was shown. The same diagram can be drawn for the Diffie Hellman on supersingular isogenies, see **??**.

$$\begin{tikzcd}
E \arrow[r, "\phi A"] \arrow[d, "\phi B"'] & E_A \arrow[d, "\phi B'"] \\
E_B \arrow[r, "\phi A'"'] & E_{AB}
\end{tikzcd}$$

Figure 1.4: Supersingular Isogeny Diffie Hellman procedure, both paths lead to the same result. Note, the different isogenies ϕ'_A and ϕ'_B, that are applied in each second step of the diagram. The reason for this lies in the mathematics of supersingular isogenies: $\phi_A \phi_B \neq \phi_B \phi_A$. In order to construct ϕ'_X or ϕ'_B, the public key of each communication partner provides additional information beside the curve E_A or E_B. **[costello2016gentle]**

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

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

2. Bob creates an isogeny ϕ_B (private key), which leads to curve E_B (part of the public key).

3. Alice and Bob exchange their public keys.

4. Bob computes ϕ'_B (using additional information from the public key of Alice) and applies ϕ'_B to the received E_A. This results in E_{AB}

5. Alice computes ϕ'_A (using additional information from the public key of Bob) and applies ϕ'_A to the received E_B. This results in E_{AB}

6. Alice and Bob now share the common secret E_{AB}.

Implemenation Details

The reference implementation of SIKE [**sike2020spec**] provides two fundamental functions: *isogen* and *isoex*. Both are used, to implement the previously introduced Supersingular Isogeny Diffie-Hellman (SIDH) algorithm. Note, that the secret key *sk* is represented as a random integer.

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

Table 1.2: Core functions of the SIKE reference implementation.

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

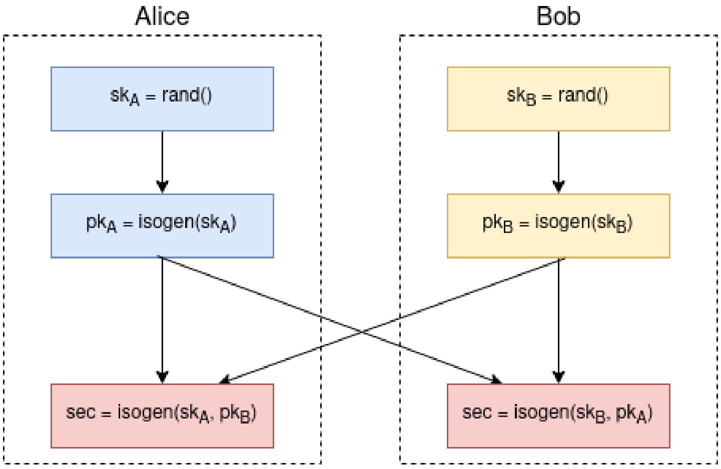

Figure 1.5: SIDH based on *isogen* and *isoex*: After generating a random secret key *sk*, each party computes its public key *pk*. After the exchange of public keys, each party finally calculates the shared secret *sec*.

Beside this key exchange algorithm, SIKE provides a complete asymmetric encryption scheme and a key encpsulation mechansim (KEM) [**sike2020spec**]. Both of these schemes build upon the here described SIKE core functions *isogen* and *isoex*.

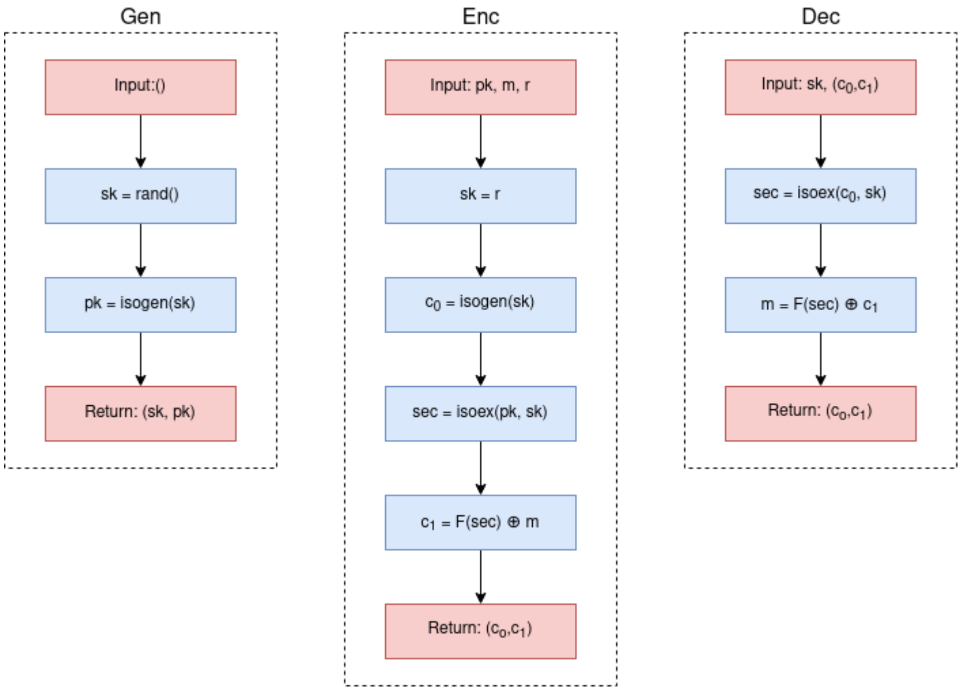

Figure 1.6: The SIKE public key encryption system consists of three algorithms:
1. *Gen* generates a key pair *(sk, pk)*.
2. *Enc* encrypts a given plaintext m using a foreign public key *pk* and the own secret key r.
3. *Dec* decodes a given cipthertext using the own secret key *sk*.
Note, that the function F used in *Enc* and *Dec* is a key derivation function.

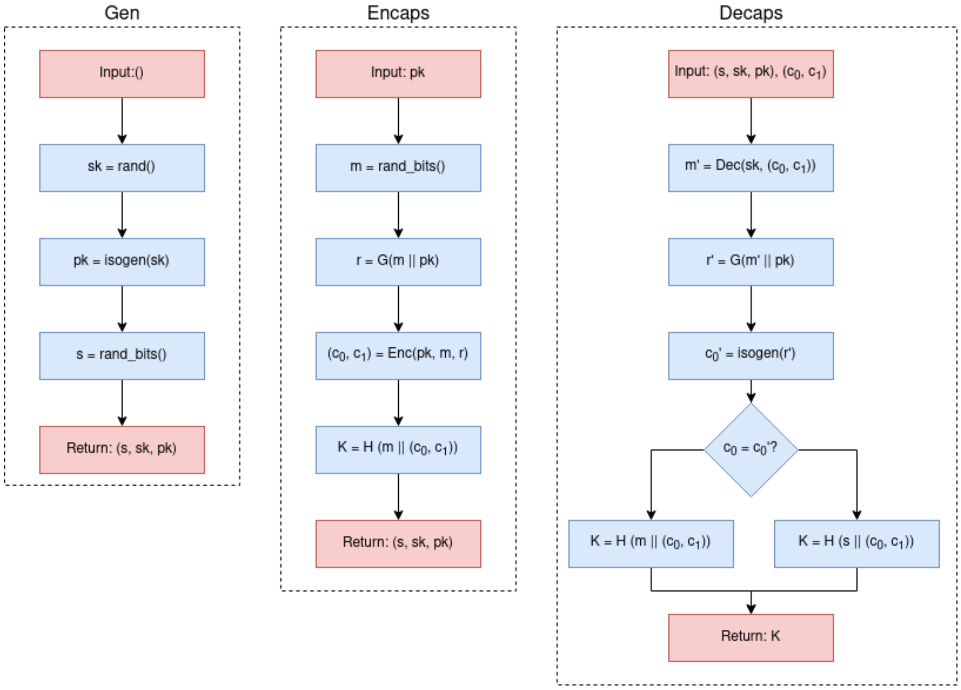

Figure 1.7: The SIKE key exchange mechanism consists of three algorithms:

1. *Gen* TODO
2. *Encaps* TODO
3. *Decaps* TODO

Security considerations

The security of SIKE is based on the hardness of the *SIDH problem*: Given two supersingular elliptic curves E and E', find an isogeny between them [**sike2020spec**].
The SIKE reference implementation proposes different parameter sets, each supposing to ensure a NIST defined security level. These NIST defined security levels are:

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

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

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

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

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

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

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

Current research provides confidence, that the SIKE parameter sets satisfy the defined security levels even under the assumption of currently known algorithms [**jaques2019quantum**]. Therefore, the authors consider three algorithms to solve the *SIDH problem*: *Tani's quantum claw finding algorithm* [**tani2009claw**], *Grover's algorithm* [**grover1996fast**] and a *parallel collision-finding algorithm* [**van1999parallel**].

Ephemeral SIDH keys are predestined to implement quantum-secure perfect forward secrecy protocols (PFS) [**longa2018note**]. PFS ensures, that a compromised long-term key does not reveal past or future keys of the protocol.

Side-channel attacks against isogeny-basd cryptography might 1) reveal parts of the secret private key or 2) reveal parts of the public key computation. To protect against power-analysis side-channel attacks it is recommend to prefer constant-time implementations [**sike2020spec**]. The authors state, however, that an attacker has *"access to a wide range of power, timing, fault and various other side-channels"*. Thus, preventing isogeny-based cryptography from all side-channel attacks seems to be nearly impossible.

2 Description of existing SIDH implementations

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

2.1 SIKE

SIKE stands for **S**upersingular **I**sogeny **K**ey **E**ncapsulation. It is the reference implementation of the first proposed isogeny-based cryptographic primitives [**jao2011towards**]. Today, SIKE is a NIST candidate for quantum-resist *"Public-key Encryption and Key-establishment Algorithms"*. It is developed by a cooperation of researchers, lead by David Jao [**sike2020spec**].
SIKE implements its key encapsulation mechanism (KEM) upon a public key encryption system (PKE), which is built upon SIDH (as highlighted in **??**). Beside a generic reference implementation, SIKE offers various optimized implementations of their cryptographic primitves:

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

All of these implementations can be run with the following parameter sets: `p434`, `p503`, `p610` and `p751`. SIKE states to countermeasure timing and cache attacks by implementing constant time cryptography [**sike2020spec**].

2.2 PQCrypto-SIDH

PQCrypto-SIDH is a software library mainly written in C. It is developed by Microsoft for experimental purposes [**microsoft2020sidh**]. Note, that many developers of SIKE also work for Microsoft leading to great similarities between SIKE and PQCrypto-SIDH. However, in terms of compression, SIKE references the here described Microsoft library in its documentation. The PQCrypto-SIDH library implements a isogney-based KEM and the underlying SIDH. Moreover, the library offers the following optimized versions:

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

All of these implementations can be run with the following parameter sets: `p434`, `p503`, `p610` and `p751`. The developers argue to protect the algorithms against timing and cache attacks. Therefore, the library implements constant time operations on secret key material [**microsoft2020sidh**].

2.3 CIRCL

CIRCL (**C**loudflare **I**nteroperable, **R**eusable **C**ryptographic **L**ibrary) is a by Cloudflare developed collection of cryptographic primitives [**circl2020github**]. CIRCL is written in Go and implements some quantum-secure algorithms like SIDH and an isogeny-based KEM. Cloudflare does not guarantee for any security within their library. Furthermore, the isogeny-based cryptographic primitives are adopted from the official SIKE implementation. The following implementation optimizations are 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 [**circl2019intro**].

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

Table 2.1: Overview and comparison of existing SIDH implementations.

3 Benchmarks

In this chapter the SIDH implementations introduced in **??** are compared with each other. For this purpose, a benchmarking suite was developed, which allows the generation of comparable benchmarks between these implementations. To get a better understanding of the results, the chapter starts with the tools and methodology used for benchmarking. The following implementations details provide precise internals of the benchmarking suite. This might be used for further development of the software. Finally, the benchmarking results are listed.

3.1 Benchmarking Methodology

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

In the following, the process of benchmarking is described within five steps. In a short version, the required steps are:

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

Create the benchmarking code

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

Compile the benchmarking code

Once the benchmarking source code is created, it needs to be compiled to a binary. Therefore, all required dependencies (libraries, headers, sourc code, ..) need to be provided to the compiler. The benchmarking suite provides a `Makefile` for each implementation, where the compilation process is implemented. All compilations performed for the benchmarking suit must use consistent compiler optimizations. This ensures the comparability of the outputs. Currently, the optimization flag `-O3` is passed to the gcc compiler. Note, that CIRCL is implemented in GO and therefore the go compiler is used for compilation. Since this compiler does not provide optimization flags, the default compiler optimizations are used [**gowiki2020compiler**]. To be able to extract benchmarks for specific functions *inlining* is disabled during compilation. The C programming language offers the following directive to disable *inlining* for a function:

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

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

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

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

Run *Callgrind*

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

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

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

Run *Massif*

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

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

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

Collect benchmarks

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

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

This information is finally used by the benchmarking suite, to produce graphs and tables for further investigation.

To further increase the quality and reproducibility of the results, the cache of the virtual operating system is cleared, before *Callgrind* and *Massif* are executed. This is done via:

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

3.2 Application Details

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

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

  - PQCrypto-SIDH x64 optimized and compressed implementation (`Microsoft_x64_Compressed`)

- CIRCL working on: `p434, p503, p610, p751`
  - CIRCL x64 optimized implementation (`CIRCL_x64`)

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

1. `secp256r1` (openssl: `prime256v1` [**turner2009elliptic**]): 128 bit security matching `SIKEp434` [**brown2010sec**]
2. `secp384r1` (openssl: `secp384r1`): 192 bit security matching `SIKEp610` [**brown2010sec**]
3. `secp512r1` (openssl: `secp521r1`): 256 bit security matching `SIKEp751` [**brown2010sec**]

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

3.2.1 Application Flow

This sections illustrates the application flow of the benchmarking suite (see **??**). Once triggered to run benchmarks, the following procedure is repeated for every implementation: The suite first compiles the benchmarking code. The binary is then executed multiple times to generate N benchmarking results, respectively for *Callgrind* and *Massif*.
Finally, the results are visualized in different formats. Graphs compare the recorded instruction counts and the peak memory consumption among implementations instantiated with comparable security classes. A HTML table lists detailed benchmarks and the Latex output is used for this document.

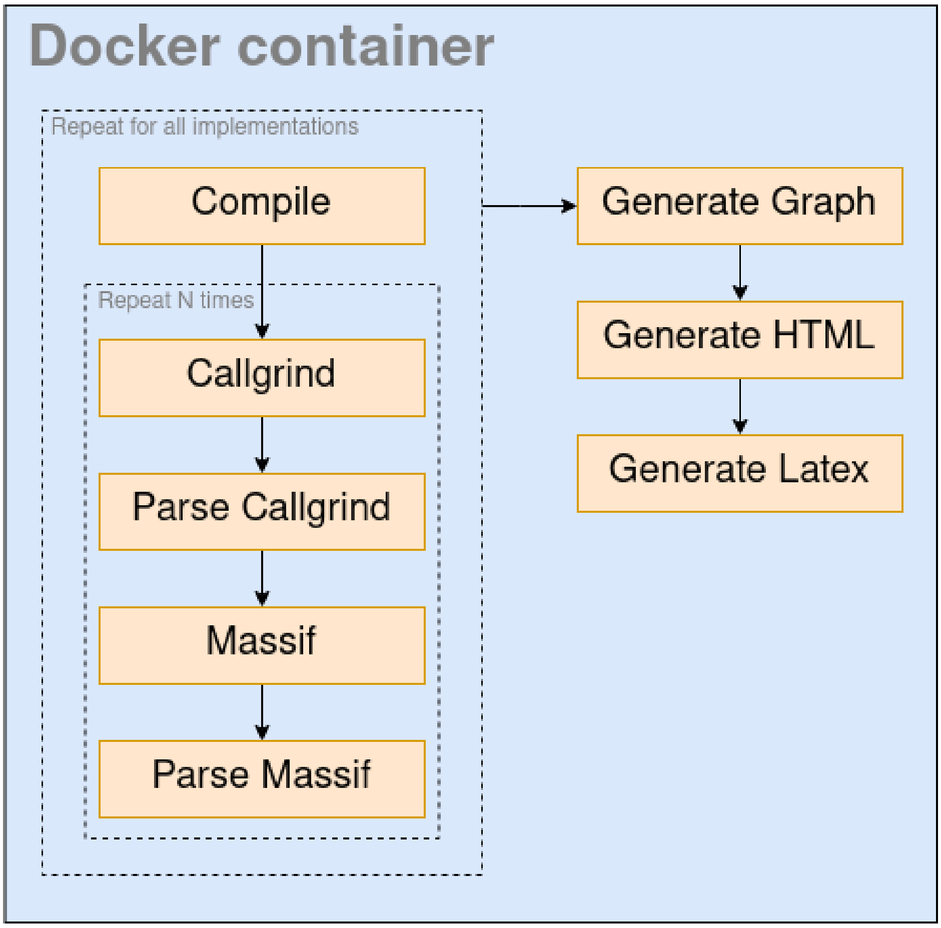

Figure 3.1: Flow chart of the benchmarking suite. For each SIDH implementation, the source code is compiled and benchmarked multiple times. The benchmarking results are visualized in different format: Graphs, HTML and Latex.

3.2.2 Application Structure

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

Representation of concrete implementations

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

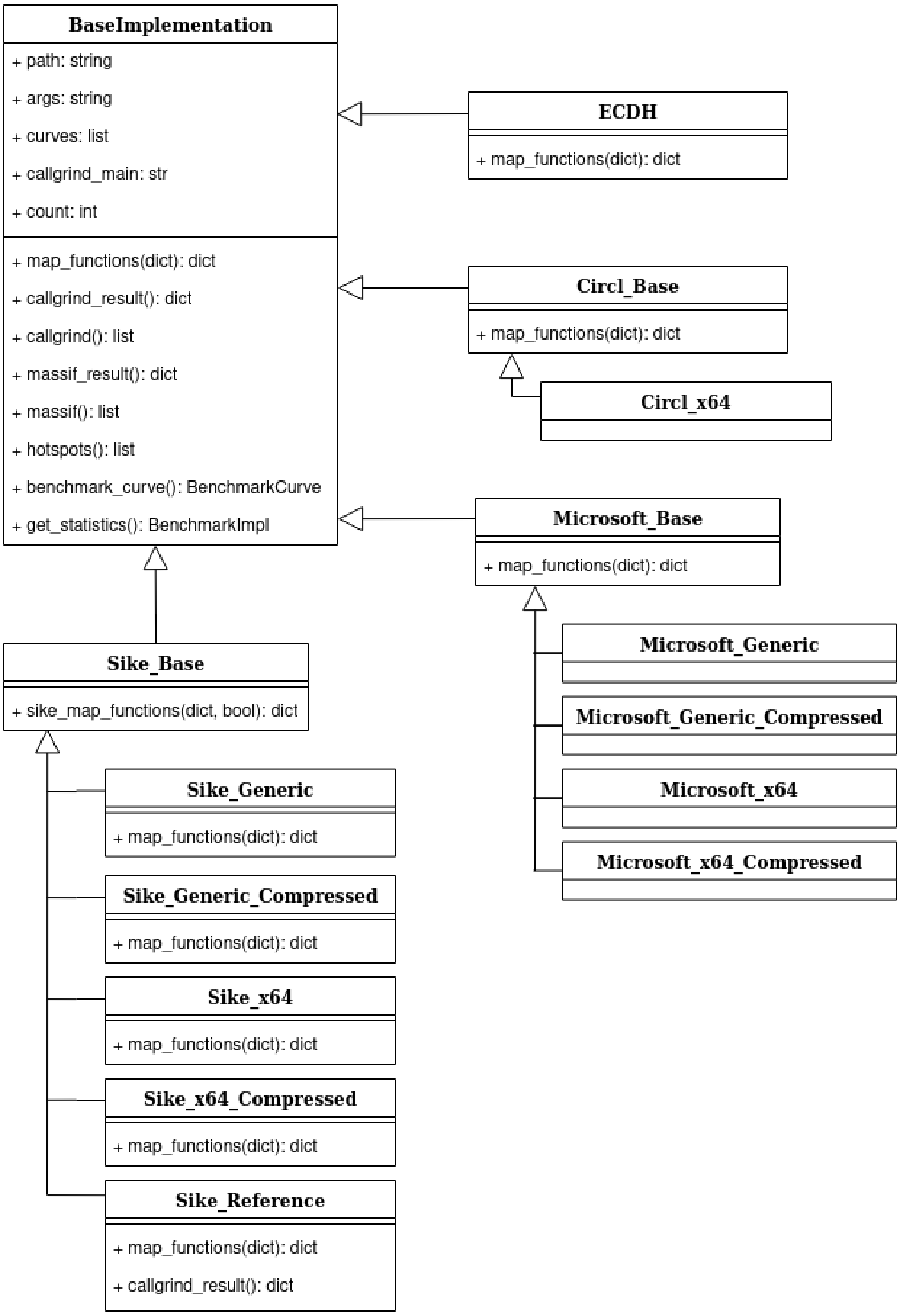
BaseImplementation
+ path: string
+ args: string
+ curves: list
+ callgrind_main: str
+ count: int
+ map_functions(dict): dict
+ callgrind_result(): dict
+ callgrind(): list
+ massif_result(): dict
+ massif(): list
+ hotspots(): list
+ benchmark_curve(): BenchmarkCurve
+ get_statistics(): BenchmarkImpl
ECDH
+ map_functions(dict): dict
Circl_Base
+ map_functions(dict): dict
Circl_x64
Microsoft_Base
+ map_functions(dict): dict
Microsoft_Generic
Microsoft_Generic_Compressed
Microsoft_x64
Microsoft_x64_Compressed
Sike_Base
+ sike_map_functions(dict, bool): dict
Sike_Generic
+ map_functions(dict): dict
Sike_Generic_Compressed
+ map_functions(dict): dict
Sike_x64
+ map_functions(dict): dict
Sike_x64_Compressed
+ map_functions(dict): dict
Sike_Reference
+ map_functions(dict): dict
+ callgrind_result(): dict

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

Representation of benchmarking results

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

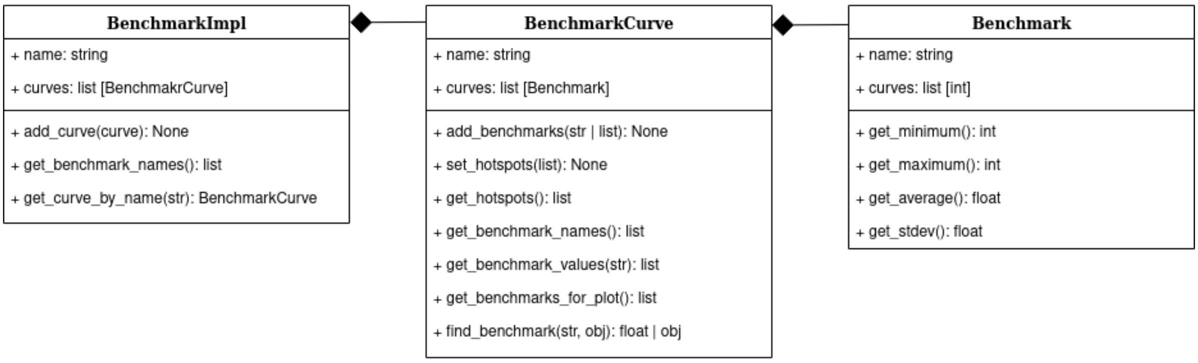
BenchmarkImpl
+ name: string
+ curves: list [BenchmakrCurve]
+ add_curve(curve): None
+ get_benchmark_names(): list
+ get_curve_by_name(str): BenchmarkCurve
BenchmarkCurve
+ name: string
+ curves: list [Benchmark]
+ add_benchmarks(str | list): None
+ set_hotspots(list): None
+ get_hotspots(): list
+ get_benchmark_names(): list
+ get_benchmark_values(str): list
+ get_benchmarks_for_plot(): list
+ find_benchmark(str, obj): float | obj
Benchmark
+ name: string
+ curves: list [int]
+ get_minimum(): int
+ get_maximum(): int
+ get_average(): float
+ get_stdev(): float

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

3.2.3 Adding Implementations

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

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

3.3 Usage

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

- Building the docker container:

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

- Running unit tests within the docker container.

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

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

- Running benchmarks within the docker container.

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

  This command triggers the *benchmarking.py* script within the docker container. This script contains the entry logic of the benchmarking suite. It benchmarks all available implementations and generates different output formats (graphs, html, latex). Once the benchmarks are done, all output files can be inspected in the folder `data/` of your current directory. The output files are:

  – *XXX.png*: These files compare the absolute instruction count of all implementations, which were run on the XXX parameter set (XXX $\in 434, 503, 610, 751$). Additionally, it contains a ECDH benchmark for comparison.

  – *XXX_mem.png*: These files compare the peak memory consumption of all implementations, which were run on the XXX parameter set (XXX $\in 434, 503, 610, 751$). Additionally, it contains a ECDH benchmark for comparison.

  – *cached.json*: This cache file contains all benchmarking results. It can be used as input for the benchmarking suite to use cached data instead of generating all benchmarks again. Since the benchmarking suite runs multiple hours if each implementation is evaluated N=100 times, this functionality provides great speed up if only the output data should change. To use the cached file as input, copy it to `container/.cached/cached.json` and run the benchmarking suite again.

  – *results.html*: This file lists detailed benchmarking results in a human readable HTML table.

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

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

3.4 Results

4 Performance Analysis

4.1 Measures for Efficiency

4.2 Security Considerations

5 Conclusion

6 Introduction

Use with pdfLaTeX and Biber.

6.1 Section

Citation test (with Biber) [**latex**].

6.1.1 Subsection

See **??**, **??**, **??**, **??**, **??**, **??**.

Table 6.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

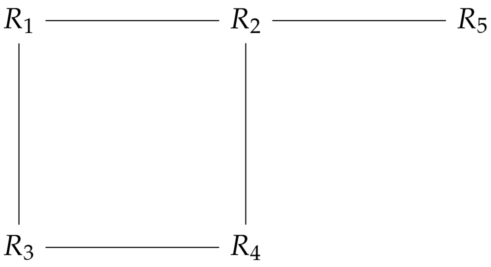

Figure 6.1: An example for a simple drawing.

This is how the glossary will be used.

Donor dye, ex. Alexa 488 ($\mathrm{D}_{\mathrm{dye}}$), Förster distance, Förster distance (R_0), and k_{DEAC}. Also, the TUM has many computers, not only one Computer. Subsequent acronym usage will only print the short version of Technical University of Munich (TUM) (take care of plural, if needed!), like here with TUM, too. It can also be –> hidden[1] <–.

[(TODO: Now it is your turn to write your thesis.
This will be a few tough weeks.)]

[(DONE: NEVERTHELESS, CELEBRATE IT WHEN IT IS DONE!)]

[1] Example for a hidden TUM glossary entry.

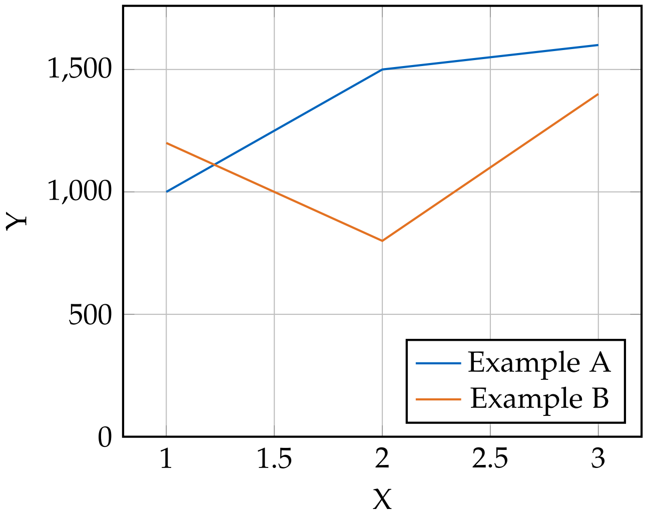

Figure 6.2: An example for a simple plot.

```sql
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 6.3: An example for a source code listing.

Figure 6.4: Includegraphics searches for the filename without extension first in logos, then in figures.

Figure 6.5: For pictures with the same name, the direct folder needs to be chosen.

(a) The logo.

(b) The famous slide.

Figure 6.6: Two TUM pictures side by side.

A General Addenda

A.1 Detailed Benchmarks

A.1.1 Benchmarks for ECDH

Parameter	secp256	secp384	secp521
PrivateKeyA	0 (0)	0 (0)	0 (0)
PublicKeyA	159.418 (0)	159.418 (0)	159.418 (0)
PrivateKeyB	0 (0)	0 (0)	0 (0)
PublicKeyB	114.430 (0)	114.430 (0)	114.430 (0)
SecretA	652.796 (0)	652.796 (0)	652.796 (0)
SecretB	651.270 (0)	651.270 (0)	651.270 (0)
Memory	12.152 (0)	12.152 (0)	12.152 (0)

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

Execution hotspots parameter *secp521*:

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

A.1.2 Benchmarks for Sike Reference

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

Table A.2: Benchmarks for Sike Reference

Execution hotspots parameter *434*:

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

Execution hotspots parameter *503*:

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

Execution hotspots parameter *610*:

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

Execution hotspots parameter *751*:

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

A.1.3 Benchmarks for Sike Generic

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

Table A.3: Benchmarks for Sike Generic

Execution hotspots parameter *434*:

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

Execution hotspots parameter *503*:

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

Execution hotspots parameter *610*:

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

Execution hotspots parameter *751*:

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

A.1.4 Benchmarks for Sike Generic Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	107 (0)
PublicKeyA	507.955.664 (0)	769.892.382 (0)	1.820.164.702 (0)	2.510.000.545 (0)
PrivateKeyB	185 (0)	145 (0)	215 (0)	200 (0)
PublicKeyB	431.555.333 (0)	644.590.547 (0)	1.209.342.668 (0)	2.160.772.067 (0)
SecretA	179.835.371 (0)	276.747.068 (0)	575.399.356 (0)	917.374.629 (0)
SecretB	225.926.297 (0)	336.288.536 (0)	644.892.098 (0)	1.128.772.498 (0)
Memory	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`: 58.79%
2. `rdc_mont`: 32.14%
3. `fp2mul434_mont`: 4.14%

Execution hotspots parameter *503*:

1. `mp_mul`: 58.4%
2. `rdc_mont`: 33.45%
3. `fp2mul503_mont`: 3.71%

Execution hotspots parameter *610*:

1. `mp_mul`: 59.39%
2. `rdc_mont`: 33.38%
3. `fp2mul610_mont`: 3.59%

Execution hotspots parameter *751*:

1. `mp_mul`: 60.47%
2. `rdc_mont`: 33.27%
3. `fp2mul751_mont`: 3.1%

A.1.5 Benchmarks for Sike x64

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

Table A.5: Benchmarks for Sike x64

Execution hotspots parameter *434*:

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

Execution hotspots parameter *503*:

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

Execution hotspots parameter *610*:

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

Execution hotspots parameter *751*:

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

A.1.6 Benchmarks for Sike x64 Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	107 (0)
PublicKeyA	49.081.389 (0)	72.924.289 (0)	124.731.117 (0)	177.153.088 (0)
PrivateKeyB	142 (0)	144 (0)	179 (0)	188 (0)
PublicKeyB	41.639.764 (0)	56.327.210 (0)	93.860.650 (0)	152.724.615 (0)
SecretA	17.064.280 (0)	23.745.225 (0)	44.070.006 (0)	63.695.110 (0)
SecretB	21.186.310 (0)	29.577.143 (0)	48.761.150 (0)	79.202.705 (0)
Memory	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`: 23.13%
3. `fpsub434`: 4.21%

Execution hotspots parameter *503*:

1. `mp_mul`: 47.93%
2. `rdc_mont`: 26.79%
3. `fpsub503`: 4.11%

Execution hotspots parameter *610*:

1. `mp_mul`: 49.73%
2. `rdc_mont`: 28.29%
3. `fpsub610`: 3.75%

Execution hotspots parameter *751*:

1. `mp_mul`: 51.7%
2. `rdc_mont`: 27.77%
3. `fpsub751`: 3.83%

A.1.7 Benchmarks for CIRCL x64

Parameter	434	503	751
PrivateKeyA	671 (0)	671 (0)	671 (0)
PublicKeyA	27.166.852 (0)	30.574.116 (0)	92.507.067 (0)
PrivateKeyB	672 (0)	672 (0)	672 (0)
PublicKeyB	28.959.882 (0)	32.771.222 (0)	103.101.279 (0)
SecretA	21.087.575 (0)	23.925.285 (0)	75.131.907 (0)
SecretB	24.364.078 (0)	27.627.558 (0)	87.853.843 (0)
Memory	21.944 (0)	22.552 (0)	22.664 (0)

Table A.7: Benchmarks for CIRCL x64

Execution hotspots parameter *434*:

1. `p434.mulP434:` 47.29%
2. `p434.rdcP434:` 22.94%
3. `p434.subP434:` 5.81%

Execution hotspots parameter *503*:

1. `p503.mulP503:` 41.41%
2. `p503.rdcP503:` 23.06%
3. `p503.subP503:` 8.4%

Execution hotspots parameter *751*:

1. `p751.mulP751:` 56.17%
2. `p751.rdcP751:` 21.72%
3. `p751.subP751:` 6.02%

A.1.8 Benchmarks for Microsoft Generic

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

Table A.8: Benchmarks for Microsoft Generic

Execution hotspots parameter *434*:

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

Execution hotspots parameter *503*:

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

Execution hotspots parameter *610*:

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

Execution hotspots parameter *751*:

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

A.1.9 Benchmarks for Microsoft Generic Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	105 (0)
PublicKeyA	366.742.854 (0)	516.734.423 (0)	1.015.885.701 (0)	1.975.978.300 (0)
PrivateKeyB	239 (0)	233 (0)	206 (0)	314 (0)
PublicKeyB	340.757.086 (0)	518.426.765 (0)	956.292.705 (0)	1.791.584.371 (0)
SecretA	177.457.712 (0)	274.004.587 (0)	569.195.809 (0)	908.464.384 (0)
SecretB	201.754.449 (0)	309.530.308 (0)	577.402.232 (0)	1.043.915.539 (0)
Memory	69.576 (0)	89.896 (0)	134.600 (0)	193.336 (0)

Table A.9: Benchmarks for Microsoft Generic Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 59.8%
2. `rdc_mont`: 32.56%
3. `fp2mul434_mont`: 4.01%

Execution hotspots parameter *503*:

1. `mp_mul`: 59.27%
2. `rdc_mont`: 33.82%
3. `fp2mul503_mont`: 2.91%

Execution hotspots parameter *610*:

1. `mp_mul`: 60.59%
2. `rdc_mont`: 33.18%
3. `fp2mul610_mont`: 3.59%

Execution hotspots parameter *751*:

1. `mp_mul`: 61.4%
2. `rdc_mont`: 33.22%
3. `fp2mul751_mont`: 2.27%

A.1.10 Benchmarks for Microsoft x64

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

Table A.10: Benchmarks for Microsoft x64

Execution hotspots parameter *434*:

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

Execution hotspots parameter *503*:

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

Execution hotspots parameter *610*:

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

Execution hotspots parameter *751*:

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

A.1.11 Benchmarks for Microsoft x64 Compressed

Parameter	434	503	610	751
PrivateKeyA	97 (0)	95 (0)	99 (0)	105 (0)
PublicKeyA	33.203.258 (0)	49.727.822 (0)	76.667.900 (0)	115.234.119 (0)
PrivateKeyB	149 (0)	152 (0)	187 (0)	200 (0)
PublicKeyB	30.783.404 (0)	41.916.959 (0)	71.642.291 (0)	118.819.557 (0)
SecretA	15.538.172 (0)	21.689.456 (0)	42.098.078 (0)	60.202.480 (0)
SecretB	17.949.480 (0)	24.857.647 (0)	42.885.843 (0)	69.856.043 (0)
Memory	69.960 (0)	90.640 (0)	135.216 (0)	193.872 (0)

Table A.11: Benchmarks for Microsoft x64 Compressed

Execution hotspots parameter *434*:

1. `mp_mul`: 52.74%
2. `rdc_mont`: 23.26%
3. `0x00000000000247dc`: 3.53%

Execution hotspots parameter *503*:

1. `mp_mul`: 49.6%
2. `rdc_mont`: 27.74%
3. `0x0000000000026694`: 3.38%

Execution hotspots parameter *610*:

1. `mp_mul`: 51.51%
2. `rdc_mont`: 29.43%
3. `0x000000000002e046`: 3.11%

Execution hotspots parameter *751*:

1. `mp_mul`: 53.61%
2. `rdc_mont`: 28.98%
3. `0x0000000000032f6d`: 2.89%

List of Figures

List of Tables