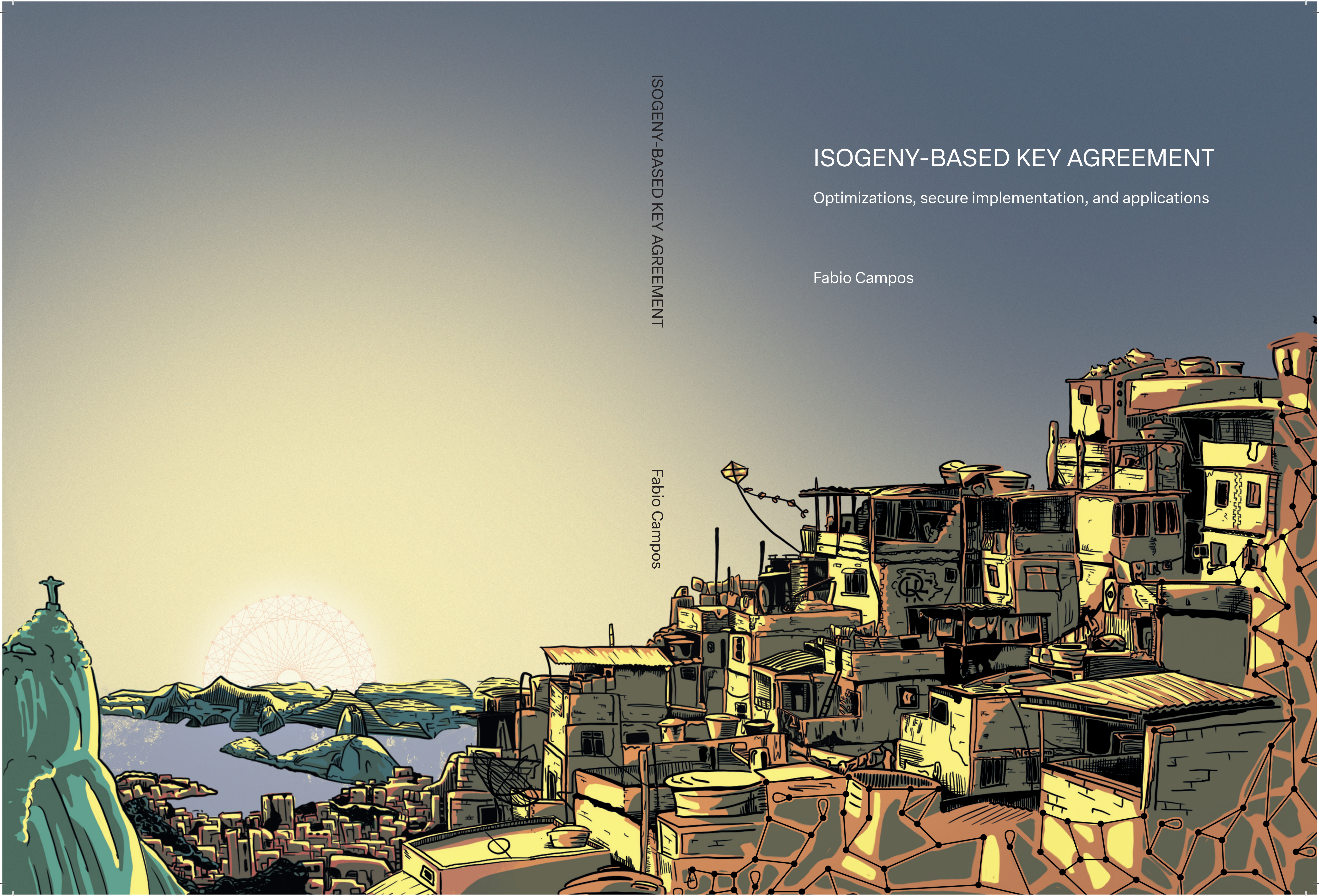


ISOGENY-BASED KEY AGREEMENT

Fabio Campos

ISOGENY-BASED KEY AGREEMENT

Optimizations, secure implementation, and applications

Fabio Campos

Isogeny-based key agreement

Optimizations, secure implementation, and applications

Fabio Campos

Email: campos@sopmac.de
Website: `https://www.sopmac.org/`

First edition July 6, 2023
Version: July 6, 2023

Printed by `https://www.proefschriften.nl/`

Cover design: Lua Maria Campos
The cover shows the *rapid-mixing* property of life exemplified by the wonderful city of Rio de Janeiro, a purgatory of beauty and chaos.

*“É muito fácil falar de coisas tão belas
De frente pro mar, mas de costas pra favela.”* – Marcelo D2 and B Negão

Isogeny-based key agreement

Optimizations, secure implementation, and applications

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

vrijdag 22 september 2023
om 12.30 uur precies

door

Fabio Felipe Campos
geboren op 16 februari 1975
te Rio de Janeiro, Brazilië

Acknowledgements

If I were to thank everyone who has helped me along this way, the list would be almost endless. For this reason, I apologize to those who are not mentioned on this list.

First of all, thanks to my supervisors Steffen Reith and Peter Schwabe for their constant support, for all the advice and encouragement during these past years.

Thanks to Steffen Reith, for dangling a carrot in front of my nose already during my Master's programme and thus showing me the way into the world of number theory. Thanks for believing in me and giving me the opportunity. Without you, all this would never have taken place. Thank you, Steffen!

Thanks to Peter Schwabe, who welcomed me with open arms from the first second on. Thank you for your patience and countless hours on e.g. assembly programming. I am really honored that you agreed to be my supervisor. Thank you, Peter!

Thanks to the members of my reading committee Chloe Martindale, Diego Aranha, Jörn Steuding, Lejla Batina, and Tim Güneysu for reviewing this thesis and providing me their valuable comments. It is a great honor to have each one of you as part of the committee.

Thanks to (*Captain*) Luca De Feo and Simona Samardjiska for being part of my defense committee.

Thanks to my *Doktorbruder* Michael Meyer for the wonderful and fruitful time with you in the office, for the countless math lessons, and for making the trips to conferences and summer schools nice and very funny. Without you this adventure would not have been so enjoyable and so successful.

Thanks to Marc Stöttinger for the countless fruitful discussions, advice, collaboration, and Spaten.

Thanks, Matthias Kannwischer, for your support and for being my inspiration in terms of effectiveness and determination during my PhD.

Thanks, Krijn Reijnders, for the fruitful collaborations and for making the last period of my PhD journey extremely enjoyable.

Thanks to Thorsten Knoll, for your support, our frequent *random oracles* meetings, and for making our research group in Wiesbaden feels more like a group.

Thank to Bo-Yin Yang for invinting me to visit Academia Sinica and the wonderful country of Taiwan.

Thanks to Krijn Reijnders, Lorenz Panny, Marc Stöttinger, Matthias Kannwischer, Michael Meyer, and Viola Campos for taking the time to proof-read parts of this thesis.

It was a great pleasure to be able to work with so many people from around the world. Thank you to my co-authors Benoît Viguier, Daan Sprenkels, Daniel J. Bernstein, Francisco Rodríguez-Henríquez, Gustavo Banegas, Hiroshi Onuki, Jana Sotáková, Jesús-Javier Chi-Domínguez, Jorge Chavez-Saab, Juliane Krämer, Krijn Reijnders, Lars Jellema, Lars Müller, Marcel Müller, Marc Stöttinger, Matthias J. Kannwischer, Mauk Lemmen, Michael Meyer, Philipp Muth, Simona Samardjiska, Steffen Reith, Steffen Sanwald, Tanja Lange, Tim Kohlstadt, Thomas Aulbach, Thom Wiggers, Tung Chou, and Yi Wang.

Thanks to Gustavo Banegas, Matthias Kannwischer, and Michael Meyer for taking time and being part of our joint lecture on Post-Quantum Cryptography.

Thanks to all my *latinos* and I am really proud being part of `criptolatino.org`.

Thanks to the people in Wiesbaden. In particular, thanks to all the people in our research group and special thanks to Carlos dos Santos, Daniel Kloos, and Thorsten Knoll for their continuous support.

Thanks to Continental AG for funding the first 3 years of my PhD position.

For my time in Bochum, I want to thank all people at the Max Planck Institute for Security and Privacy. It was a relatively short but very nice time.

Thank you to the all people at the Radboud University. I always feel very welcome.

Muito obrigado à minha família e amigos no Brasil. Apesar da distância, é bom saber que vocês estão presentes.

I want to thank my friends, family, and especially my lovely sister Handrea because without her help and love, my life would definitely have been different and I definitely would not have achieved this.

Danke an Erich und Sonja für die Gastfreundschaft und die sehr liebvolle Verpflegung während schwieriger Phasen in den letzten Jahren.

Thanks to my lovely daugthers, Lua and Malu, for your patience, for understanding my bad mood and mental absence during certain projects, and sorry for the boring technical discussions at the dining table.

Thanks Viola for her constant love, support, and motivation, as well as for the countless times technical discussions and help in the last few years. Without you, I would never have got this far.

And finally, I thank one of the most special persons I met in my life, my mother, who planted the seed for my path, even though she left me too soon.

Contents

1 Introduction

"Imagine there's no crypto";[1] even if John Lennon were alive and wrote the song *Imagine* [126] today, it is unlikely that it would contain this phrase. Although the current world without cryptography is at least as unimaginable as a world without possessions or countries. *Cryptography* is the science or even the art of designing protocols and algorithms for protecting data, and its long and fascinating history dates back to some limited use by the Egyptians about 4000 years ago, according to Kahn [113]. Among many other examples, such as the art of secret writing saving Greece from being conquered by the Persians in the fifth century B.C. [173], the *Caesar cipher*, and the *Enigma* encryption machine [113] during World War II, all of these methods are based on *symmetric cryptography.* Symmetric cryptography, also known as *single-key*, *private-key* or *secret-key* cryptography, is the use of a single shared secret to encrypt data to safely share it between parties. Prior to the introduction of *public-key* or *asymmetric* cryptography, any secret information used in encoding secret messages had to be shared in a private channel before being able to exchange messages. In 1976 [77], Diffie and Hellman[2] proposed public-key cryptography by describing how *one-way functions* and *trapdoor permutations* can be used to build cryptographic protocols. While secret-key cryptography requires both parties to share a secret, public-key cryptography uses a pair of keys. Each such key pair consists of a *public key* and a *private key*, with the requirement that it should not be computationally feasible to derive the private key from the public key. Public-key cryptography allows building basic mechanisms: public-key encryption (PKE), non-interactive key-exchange (NIKE), key-encapsulation mechanisms (KEMs), and digital signatures, which essentially form the basis of modern digital communication.

The security of these primitives inherently relies on the computational hardness of the underlying problems. Essentially all public-key protocols widely deployed today rely on the hardness of the integer factorization prob-

[1] "crypto" means cryptography.

[2] Although the invention of public-key cryptography is often attributed to Diffie and Hellman, we refer to "The Alternative History of Public Key Cryptography" in [173].

lem (RSA [162]), the discrete-logarithm problem (Diffie and Hellman [77], ElGamal [93]), or the elliptic-curve discrete-logarithm problem (Miller [139] and Koblitz [115]).

Unfortunately, due to Shor's algorithm [170] these problems can be efficiently solved by a quantum computer. Once sufficiently-large quantum computers are available, they will be able to break cryptographic schemes building upon these problems. Apart from being able to decrypt encrypted messages and forge digital signatures, even today an attacker can store encrypted communication and decrypt it retroactively once a quantum computer is available. Due to long-term data storage, this represents a realistic risk. Further, the transition to a *post-quantum era* is a complex process [144]: It requires, e.g., the development and deployment of hardware and software solutions, the establishment of standards, and the migration of systems. Thus, according to Lange in [29]: "It is not about developing something we want to use in ten or 15 years, but something we would need to use now."

We point out that symmetric schemes are also subject to quantum attacks. However, Grover's quantum algorithm [101] only yields a quadratic speedup for key searches, which means that compared to classical security, doubling the key lengths results in comparable post-quantum security.

Due to the dramatic impact on asymmetric cryptography, in 2016 the United States National Institute of Standards and Technology (NIST)[3] started a public standardization process [152] for post-quantum cryptography (PQC), where researchers were asked to submit supposedly quantum-resistant schemes. The goal of this project is to standardize cryptographic algorithms that are secure against classical and quantum computers. Specifically, these algorithms should serve as replacements for key establishment (NIST SP 800-56A [149] and NIST SP 800-56B [150]), and digital signatures (NIST FIPS186-4 [148]).

NIST's standardization process consists of several rounds, where schemes were eliminated with respect to certain evaluation criteria in each round. From 69 schemes in the first round, only 15 schemes (8 KEMs, 7 signature schemes) made it to the third round.

Among those submissions the following approaches are being pursued to realize post-quantum cryptography: code-based, hash-based, isogeny-based, lattice-based, and multivariate-based cryptography. Since these approaches are based on different assumptions and have different performance profiles, they offer both disadvantages and advantages.

After three rounds, on July 05, 2022 NIST selected[4] CRYSTALS-KYBER [9] (key establishment) and CRYSTALS-DILITHIUM [131], Falcon [160], and SPHINCS+ [8] (digital signatures) for standardization. However, NIST also plans to standardize further schemes for key-establishment

[3]`https://www.nist.gov/`

[4]`https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`

in the fourth round[5] and to start a new call for proposals for digital-signature algorithms during 2022.

Isogeny-based cryptography, which forms the central topic of this thesis, is the youngest of the five PQC approaches, and is based on the hardness of finding isogenies between elliptic curves. Its main advantages are key-exchange protocols with the smallest key sizes among the post-quantum schemes and the only non-interactive key exchange.

Isogeny-based cryptography was first proposed by Couveignes in 1997, but only published in 2006 [66] after the independent rediscovery of his scheme by Stolbunov and Rostovtsev [164]. This scheme, denoted CRS (Couveignes–Rostovtsev–Stolbunov), did not gain much attention as the construction was rather inefficient.

In 2011, following the approach of Charles, Goren, and Lauter [53], a different isogeny-based scheme with much better performance was proposed by Jao and De Feo: Supersingular Isogeny Diffie–Hellman (SIDH) [109]. In 2016, a KEM version of SIDH was submitted to the NIST PQC process under the name SIKE [108], which proceeded to the fourth round.

Major advances in isogeny-based cryptography have been made in the last years, leading to new isogeny-based schemes. This includes the non-interactive key exchange CSIDH [49, 12], the SIDH variant B-SIDH [60], and signature schemes such as SeaSign [71], CSI-FiSh [28] and SQISign [84].

Remark 1. *At the time of writing this thesis, Castryck and Decru presented in [46] a heuristic polynomial-time key-recovery attack that breaks all proposed SIKE parameter sets. Maino and Martindale independently discovered fundamentally the same attack [132]. Some countermeasures based on masking information required for these attacks have been proposed [143, 85]. For further details on these attacks, some improvements, their implementation, and results, we refer to [163, 155].*

Although these attacks have a dramatic impact on SIDH-based schemes, the impact on the results of this thesis is minor. On the one hand, because the security of CSIDH-based schemes (see Section 4.1, Section 4.2, Section 5.1, Section 6.1, and partly Section 5.2 and Section 5.3) is currently not affected by these attacks and, on the other hand, because the SIDH-related work (see Section 3.1, and partly Section 5.2 and Section 5.3) partly dealt with low-level properties of the scheme and can thus be transferred to future protocols.

1.1 Outline and Contributions

All the work and results presented in this thesis are the result of collaborations with several co-authors. The following section provides the outline

[5]`https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions`

of this thesis and an overview of the underlying publications, thereby highlighting my contributions.

Chapter 2 surveys background underlying the following chapters. It covers elliptic curves, isogenies between elliptic curves, and cryptographic background by introducing some basic terminology, as well as an introduction to the relevant isogeny-based schemes.

Chapter 3 focuses on possible optimizations of SIDH. It investigates how to efficiently switch between Montgomery and twisted Edwards curves within SIDH, and how to integrate Edwards curve arithmetic in the SIDH implementation. This chapter is based on the following preprint:

> Michael Meyer, Steffen Reith, and Fabio Campos. On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic. Cryptology ePrint Archive, Paper 2017/1213, 2017. `https://eprint.iacr.org/2017/1213`

In this work, I primarily contributed by developing and evaluating all the C code involved in the curve arithmetic. The writing of the paper was a joint effort of all authors.

Chapter 4 introduces two approaches to evaluate the CSIDH group action in constant time. In the first section, it presents the first efficient constant-time implementation of CSIDH. Apart from introducing the usage of dummy isogenies in this context, it further presents several speedups. The second algorithm (CTIDH) introduces a new key space and a corresponding new algorithm achieving speed records for constant-time CSIDH. This chapter is based on the following two publications:

> Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*, volume 11505 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2019. `https://doi.org/10.1007/978-3-030-25510-7_17`

In this work, I implemented and evaluated the optimized constant-time algorithm in C based on the implementation presented in [49]. I was further involved in designing the optimizations. The paper was written by all authors.

> Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana

Sotáková. CTIDH: faster constant-time CSIDH. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):351–387, 2021. `https://doi.org/10.46586/tches.v2021.i4.351-387`

In this work, I was involved in the implementation and evaluation of the first proof of concept version of CTIDH at an earlier stage of the project. Further, I was in charge of dealing with the search for optimal parameters. The writing of the paper was a joint effort of all authors.

Chapter 5 focuses on physically attacking isogeny-based schemes. It evaluates how practical fault-injection attacks are on constant-time implementations of CSIDH based on dummy calculations. Then, it discusses possible countermeasures to protect against such fault injections. Further, it presents safe-error attacks against SIKE and CSIDH. For this, it demonstrates that full key recovery is possible in some scenarios by physically carrying out two of the presented attacks. Finally, it presents zero-value and correlation attacks by analyzing the behavior of the zero and six curve in CSIDH and SIKE. This chapter is based on the following three publications:

Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations Against Fault Injection Attacks. In *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, pages 57–65. IEEE, 2020. `https://doi.org/10.1109/FDTC51366.2020.00015`

In this work, I implemented all the practical attacks, including the Cortex-M4 implementation of CSIDH, and all the countermeasures. Further, I was also involved in designing the attacker models and implementing the simulations in Python. The writing of the paper was a joint effort of all authors.

Fabio Campos, Juliane Krämer, and Marcel Müller. Safe-error attacks on SIKE and CSIDH. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 11th International Conference, SPACE 2021, Kolkata, India, December 10-13, 2021, Proceedings*, volume 13162 of *Lecture Notes in Computer Science*, pages 104–125. Springer, 2021. `https://doi.org/10.1007/978-3-030-95085-9_6`

In this work, I was in charge of all the practical experiments. Further, I was also involved in elaborating the safe-error attacks. The writing of the paper was a joint effort of all authors.

> Fabio Campos, Michael Meyer, Krijn Reijnders, and Marc Stöttinger. Patient zero and patient six: Zero-value and correlation attacks on CSIDH and SIKE. In Benjamin Smith and Huapeng Wu, editors, *Selected Areas in Cryptography - SAC 2022 - 29th International Conference, Ontario, Canada, August 24-26, 2022, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2022. `https://eprint.iacr.org/2022/904`

In this work (accepted at SAC 2022), the design of all attacks was a joint effort of all authors. I was in charge of all the practical experiments on CSIDH and variants. The writing of the paper was a joint effort of all authors.

Chapter 6 presents an actively-secure threshold scheme in the setting of hard homogenous spaces allowing fine-grained access structures. This chapter is based on the following publication:

> Fabio Campos and Philipp Muth. On actively secure fine-grained access structures from isogeny assumptions. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 375–398. Springer, 2022. `https://eprint.iacr.org/2021/1109`

In this work, I primarily contributed by adapting our approach to the setting of hard homogeneous spaces and designing the signature protocols. As before, writing the paper was a joint work of both authors.

Further publications. I decided not to include some publications that either are not related to isogeny-based cryptography or were published after the review by the reading committee. The publications related to cryptography not appearing in this thesis in chronological order are the following:

> Fabio Campos, Michael Meyer, Steffen Sanwald, Marc Stöttinger, and Yi Wang. Post-quantum cryptography for ECU security use cases. In *17th escar Europe : embedded security in cars (Konferenzveröffentlichung)*. 2019. `https://doi.org/10.13154/294-6673`

> Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. LMS vs XMSS: comparison of stateful hash-based signature schemes on ARM Cortex-M4. In Abderrahmane Nitaj and Amr M. Youssef, editors, *Progress in Cryptology -*

AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings, volume 12174 of *Lecture Notes in Computer Science*, pages 258–277. Springer, 2020. `https://doi.org/10.1007/978-3-030-51938-4_13`

Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 526–545. Springer, 2020. `https://doi.org/10.1007/978-3-030-65411-5_26`

Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace. Cryptology ePrint Archive, Paper 2023/335, 2023. `https://eprint.iacr.org/2023/335` (accepted at TCHES 2023)

Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. On the practicality of post-quantum TLS using large-parameter CSIDH. Cryptology ePrint Archive, Paper 2023/793, 2023. `https://eprint.iacr.org/2023/793` (under submission)

A list of all publications including further material is available at `https://www.sopmac.org/research/`.

1.2 Data Management

The following research datasets have been produced during this PhD research and are available online:

Chapter 3.1: On hybrid SIDH schemes
`https://github.com/sopmacF/hybrid-SIDH`

Chapter 4.1: Efficient constant-time implementation of CSIDH
`https://github.com/sopmacF/On-Lions-and-Elligators`

Chapter 4.2: CTIDH: faster constant-time CSIDH
`https://github.com/sopmacF/CTIDH`

Chapter 5.1: Protecting CSIDH with Dummy-Operations
`https://github.com/csidhfi/csidhfi`

Chapter 5.2: Safe-Error Attacks on SIKE and CSIDH
`https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC`

Chapter 5.3: Zero-Value and Correlation Attacks on SIKE and CSIDH
`https://github.com/PaZeZeVaAt/simulation`

For more details, see Research Data Management.

2 Preliminaries

This chapter presents the most important background for the following chapters of this thesis. Section 2.1 briefly introduces elliptic curves as a main building block for isogeny-based cryptography. Section 2.2 gives an introduction to the use of isogenies between elliptic curves in cryptography following [59, 70, 135, 156]. For further mathematical background on these topics, we refer to [187, 172, 89, 59, 70]. Section 2.3 presents some cryptographic background by introducing the basic terminology and the security properties of key-establishment, digital-signature, and threshold schemes. Section 2.4 introduces isogeny-based protocols relevant for this thesis such as SIDH and CSIDH. Lastly, Section 2.5 briefly discuss physical attacks by introducing the basic aspects of side-channel and fault attacks.

2.1 Elliptic Curves

The security of *elliptic-curve cryptography* (ECC) relies on the hardness of the discrete-logarithm problem in the group of points on an elliptic curve defined over a finite field. The elliptic curve discrete logarithm problem (ECDLP) is believed to be asymptotically harder than the factorization of integers or the computation of discrete logarithms in the multiplicative group of a finite field. Thus, ECC provides smaller key sizes for the same level of security. In the following, we will introduce the most basic aspects of elliptic curves required to follow the introduction to isogenies in Section 2.2.

Throughout the preliminaries, $\mathbb{K}$ refers to a field, $\overline{\mathbb{K}}$ refers to its algebraic closure, and E, E_1, E_2 refer to elliptic curves over a given field.

Definition 1 (Short Weierstraß curve)**.** *Let $\mathbb{K}$ be a field of characteristic $\notin \{2,3\}$. A short Weierstraß curve E over $\mathbb{K}$ is defined by the following equation*

$$E : y^2 = x^3 + ax + b, \tag{2.1}$$

where $a, b \in \mathbb{K}$ and $4a^3 + 27b^2 \neq 0$.

Requiring $4a^3 + 27b^2 \neq 0$ ensures the non-singularity of E.

For other curve models such as Montgomery, Edwards, and twisted Edwards curves, which offer computational advantages allowing more efficient group law operations, we refer to Section 3.1 and further to [140, 80, 17].

The set of points on E over $\mathbb{K}$ denoted by $E(\mathbb{K})$ is

$$E(\mathbb{K}) = \{\infty_E\} \cup \{(x, y) \in \mathbb{K}^2 : y^2 = x^3 + ax + b\},$$

where ∞_E is the point at infinity. Instead of working with affine coordinates, we can translate the affine curve Equation (2.1) to a projective curve defined by the equation

$$E : Y^2 Z = X^3 + aXZ^2 + bZ^3,$$

with coordinates $(X : Y : Z) \in E$ and $x = X/Z$ and $y = Y/Z$ for converting the coordinates. The point at infinity is given by $\infty_E = (0 : 1 : 0)$, which lacks an affine representation. Based on this resulting set of points, which carries a group structure, we define the additive group law [172] as follows.

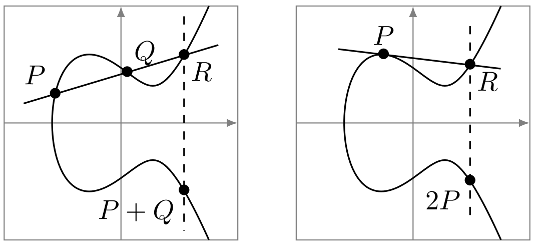
Theorem 1 (Group Law). *The addition of points on an elliptic curve E over $\mathbb{K}$ satisfies the following properties:*

- *(identity) $P + \infty_E = P$ for all points $P \in E(\mathbb{K})$.*
- *(inverse) For any $P \in E(\mathbb{K})$, there exists a unique $P' \in E(\mathbb{K})$ (commonly denoted by $-P$) such that $P + P' = \infty_E$.*
- *(closed) For any points $P, Q \in E(\mathbb{K})$, it follows $P + Q \in E(\mathbb{K})$.*
- *(associativity) For any points $P, Q, R \in E(\mathbb{K})$, we have $(P + Q) + R = P + (Q + R)$.*

For elliptic curves defined over $\mathbb{R}$, the addition of points is calculated as follows. Let P, Q be points on $E(\mathbb{R})$, and L be the line connecting P and Q (see Figure 2.1 a) or a tangent line to $E(\mathbb{R})$ in the case $P = Q$ (see Figure 2.1 b). Then, let R be the third point of intersection of L with the curve. Let L' be the line connecting R and ∞_E. Then $P + Q$ is the point where L' intersects the curve. Based on this *chord-and-tangent* addition rule and on the group law from Theorem 1, $(E(\mathbb{K}), +)$ is an *Abelian group*. We refer to [187, Section 2.2] for more details.

From now on, we will mainly focus on elliptic curves defined over finite fields. For this, we set $\mathbb{K} = \mathbb{F}_q$, such that p is prime and $q = p^n$ a power of p where $n \in \mathbb{Z}^+$. Thus, $\mathbb{K}$ is a finite field with q elements and characteristic p. The previously defined properties also apply to elliptic curves defined over finite fields.

Based on the group law, we can define a subgroup of E called *torsion subgroup* as follows.



a) “Chord rule” b) “Tangent rule”

Figure 2.1: Geometric representation of the chord-and-tangent addition rule.

Definition 2 (m-Torsion Subgroup)**.** *Let E be an elliptic curve, m a positive integer, and let $[m]$ be the (scalar-)multiplication-by-m map*

$$[m] : E \rightarrow E.$$

Then, the kernel of the map is denoted by $E[m]$. We refer to $E[m]$ as the m-torsion subgroup:

$$E[m] = \{P \in E(\overline{\mathbb{K}}) \mid [m]P = \infty_E\}.$$

Further, we can define the *order of points* as follows.

Definition 3 (Order of Points)**.** *Let $P \in E(\mathbb{K})$ be a point. Then the order of P is either the smallest integer $m > 0$ such that $[m]P = \infty_E$ or infinite, if no such integer exists.*

The order of P determines how many points the cyclic subgroup generated by P $\langle P \rangle = \{P, [2]P, [3]P, ..., \infty_E\}$ contains. Since the structural properties of an elliptic curve are strongly related to the *p–torsion* subgroup, we further define:

Definition 4 (Supersingular/Ordinary Curves)**.** *Let E be an elliptic curve defined over a field $\mathbb{K}$ of characteristic $p > 0$. If $E[p] \simeq \mathbb{Z}/p\mathbb{Z}$ then E is called ordinary, and if $E[p] \simeq \{\infty_E\}$, then E is called supersingular*[6]*.*

Since the cryptographic applications studied in this thesis work on supersingular curves, this thesis will only focus on supersingular elliptic curves. Since $\mathbb{K}$ is finite, $E(\mathbb{K})$ is also a finite group.

Theorem 2 (Hasse's Theorem)**.** *Let $E(\mathbb{K})$ be an elliptic curve over a finite field $\mathbb{K}$ with q elements. Then, the number of points $\#E(\mathbb{K})$ satisfies*

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{K}) \leq q + 1 + 2\sqrt{q}.$$

[6]The term “supersingular” is not related to the term “singular”.

2.2 Isogenies

After briefly introducing elliptic curves, we can zoom out and look at maps between elliptic curves. Since these maps are at the heart of all isogeny-based cryptographic protocols, they represent the central topic of this thesis.

Definition 5 (Isogeny)**.** *An isogeny φ is a non-constant morphism between two elliptic curves E_1 and E_2*

$$\varphi : E_1 \to E_2,$$

which maps ∞_{E_1}, the point at infinity of E_1, to ∞_{E_2}, the point at infinity of E_2,

$$\varphi(\infty_{E_1}) = \infty_{E_2}.$$

Two curves E_1, E_2 are called *isogenous* whenever there exists an isogeny $\varphi : E_1 \to E_2$, and vice versa.

Let $\mathrm{Hom}(E, E_1)$ be the set of all isogenies $E \to E_1$ over $\mathbb{K}$ together with the *zero* map.[7] Further, let $\varphi, \varphi' \in \mathrm{Hom}(E, E_1)$ and $P \in E(\mathbb{K})$, then the set $\mathrm{Hom}(E, E_1)$ together with a point-wise addition has a group structure given by $(\varphi + \varphi')(P) = \varphi(P) + \varphi'(P)$.

The multiplication-by-m map $[m] : E \to E$ (see Definition 2) that maps any point on E to its multiple, and the identity map $\varphi(P) = P$ are examples for isogenies. Since they map from E to itself, they are *endomorphisms.*

Definition 6 (Endomorphism)**.** *Let E be an elliptic curve. An endomorphism of E is an isogeny from E to itself $\varphi : E \to E$, or the zero map. In this case, we denote*

$$\varphi \in \mathrm{End}(E) = \mathrm{Hom}(E, E).$$

Along with the point-wise addition from $\mathrm{Hom}(E, E)$ and the composition of endomorphisms as multiplication, this set forms the endomorphism ring of E.

In the case of elliptic curves over finite fields, we should consider the following special endomorphism.

Definition 7 (Frobenius Endomorphism)**.** *Let E be an elliptic curve over a finite field $\mathbb{K}$. Then the map*

$$\pi : E \to E, \quad (x, y) \mapsto (x^q, y^q)$$

is called the (q^{th}-power) Frobenius endomorphism.

Isogenies can be classified depending on the relation between their degree as a rational map and the finite number of points mapped to ∞, i.e. their kernel.

[7]The *zero* map sends everything to zero.

Definition 8 (Separable Isogenies)**.** *An isogeny φ is called separable if the cardinality of the kernel of the isogeny equals the degree of the isogeny such that*

$$\#\ker(\varphi) = deg(\varphi),$$

and it is called inseparable otherwise.

In this thesis, we are mostly interested in separable isogenies.

Proposition 3. *Let E_1 be an elliptic curve and let G be a finite subgroup $G \subset E_1$, both defined over $\mathbb{K}$. Then there is an elliptic curve E_2 and a separable isogeny $\varphi_G : E_1 \to E_2$ such that $\ker(\varphi_G) = G$. Further, the pair (E_2, φ_G) is unique up to post-composition with isomorphisms.*

In this case, the curve E_2, also called *codomain* curve, is often written as E_1/G. For more details on how to compute isogenies and the resulting codomain curve, we refer to [184, 19].

Given two elliptic curves defined over a finite field, to determine whether they are isogenous, we can verify that the following applies.

Theorem 4 (Tate's Theorem [187])**.** *Two elliptic curves E_1, E_2 defined over a finite field $\mathbb{K}$ are isogenous over $\mathbb{K}$ if and only if $\#E_1(\mathbb{K}) = \#E_2(\mathbb{K})$.*

In fact, the relation of being isogenous describes an equivalence relation where the resulting equivalence classes are the set of isomorphism classes of supersingular elliptic curves. Two elliptic curves are isomorphic over $\overline{\mathbb{K}}$ if and only if they have the same *j-invariants.*

Definition 9 (j-invariant)**.** *Let E be an elliptic curve given by a Weierstrass equation $E : y^2 = x^3 + ax + b$. The j-invariant of E is defined as*

$$j(E) = 1728\left(\frac{4a^3}{4a^3 + 27b^2}\right).$$

The number of isomorphism classes is finite and can be calculated as follows.

Theorem 5 (Number of Isomorphism Classes)**.** *Let $\mathbb{K}$ be a finite field of characteristic $p > 3$, Then, there are (up to isomorphism) exactly*

$$\lfloor p/12 \rfloor + \varepsilon_p$$

supersingular elliptic curves, where $\varepsilon_p = 0, 1, 1, 2$ if $p \equiv 1, 5, 7, 11 \mod 12$, respectively.

For every isogeny $\varphi : E_1 \to E_2$ there exists a complementary isogeny in the opposite direction $\hat{\varphi} : E_2 \to E_1$, the dual isogeny.

Theorem 6 (Dual Isogeny)**.** *Let $\varphi : E_1 \to E_2$ be an isogeny. There is a unique isogeny $\hat{\varphi} : E_2 \to E_1$ such that $\hat{\varphi} \circ \varphi = [\deg \varphi]$.*

Based on the previous facts, we can look at isogenies as edges in a graph as follows.

Definition 10 (Isogeny Graph)**.** *Let $\mathbb{K}$ be a finite field, S be a set of positive integers not divisible by char$(\mathbb{K})$. We define the S-isogeny graph $G_{\mathbb{K},S}(V,E)$ over $\mathbb{K}$ as follows:*

- *The vertices V represent elliptic curves defined over $\mathbb{K}$, up to $\mathbb{K}$-isomorphisms.*
- *The edges E represent isogenies of degree $\ell \in S$ defined over $\mathbb{K}$, up to post-composition with $\mathbb{K}$-isomorphisms.*

In the case $S = \{\ell\}$, we write $G_{\mathbb{K},\ell}(V,E)$ for the ℓ-isogeny graph.

The resulting graph is undirected due to the existence of dual isogenies. The security of the resulting schemes is strongly related to the mixing properties of the underlying isogeny graph. In fact, as shown in [159], such ℓ-isogeny graphs are connected, $(\ell+1)$-regular[8], and have the *Ramanujan* property [159]. Informally, this means that any node within the graph can be reached from any other node with a small number of steps (*rapid mixing*). For more details on the rapid-mixing properties of such graphs, we refer to [110].

[8]A k-regular graph is a graph where each vertex has k neighbors.

2.3 Cryptographic Constructions

In the following sections, we introduce the terminology and the security properties of public-key protocols. For further details on the security notions, we refer to [34].

2.3.1 Key-Establishment Schemes

The main motivation of key establishment is to allow parties to agree on a shared secret usually used for subsequent communication. In the following, we introduce three approaches for setting up such a key-establishment process.

2.3.1.1 Public-Key Encryption

Definition 11 (Public-Key Encryption (PKE)). *Let M be a message space, PK be a public-key space, SK be a private-key space, and C be a ciphertext space. A public-key encryption scheme is defined by the tuple of probabilistic algorithms (`KeyGen`, `Encrypt`, `Decrypt`) as follows:*

`KeyGen`$() : \emptyset \rightarrow PK \times SK$
: *takes no inputs and outputs a key pair consisting of a public key $pk \in PK$ and a secret key $sk \in SK$.*

`Encrypt`$(pk, m) : PK \times M \rightarrow C$
: *takes as input a public key $pk \in PK$ and a message $m \in M$ and outputs a ciphertext $c \in C$ representing the encrypted message.*

`Decrypt`$(sk, c) : SK \times C \rightarrow M$
: *takes as input a secret key $sk \in SK$ and a ciphertext $c \in C$ and outputs the decrypted message $m' \in M$.*

For correctness, it is required that for any key pairs $(pk, sk) \in PK \times SK$ generated by the algorithm `KeyGen` and any message $m \in M$ the algorithms `Encrypt` and `Decrypt` satisfy the constraint

$$\texttt{Decrypt}(sk, \texttt{Encrypt}(pk, m)) = m.$$

2.3.1.2 Key-Encapsulation Mechanism

Definition 12 (Key-Encapsulation Mechanism (KEM)). *Let PK be a public-key space, SK be a private-key space, C be a ciphertext space, and S be a shared-secret space. A KEM is defined by the tuple of probabilistic algorithms (`KeyGen`, `Encaps`, `Decaps`) as follows:*

$\texttt{KeyGen}() : \emptyset \rightarrow PK \times SK$
: *takes no inputs and outputs a key pair consisting of a public key $pk \in PK$ and a secret key $sk \in SK$.*

$\texttt{Encaps}(pk) : PK \rightarrow C \times S$
: *takes as input a public key $pk \in PK$ and outputs a ciphertext $c \in C$ and a shared secret $s \in S$.*

$\texttt{Decaps}(sk, c) : SK \times C \rightarrow S$
: *takes as input a secret key $sk \in SK$ and a ciphertext $c \in C$ and outputs a shared secret $s' \in S$.*

For correctness, it is required that, for any key pairs $(pk, sk) \in PK \times SK$ generated by the algorithm `KeyGen` and given

$$(c, s) \leftarrow \texttt{Encaps}(pk)$$

the algorithms `Decaps` satisfies the constraint

$$s = \texttt{Decaps}(sk, c).$$

2.3.1.3 Non-Interactive Key Exchange

Definition 13 (Non-Interactive Key-Exchange (NIKE))**.** *Let PK be a public-key space, SK be a private-key space, and S be a shared-secret space. A NIKE protocol is defined by the tuple (`KeyGen`, `SharedKey`) as follows:*

$\texttt{KeyGen}() : \emptyset \rightarrow PK \times SK$
: *takes no inputs and outputs a key pair consisting of a public key $pk \in PK$ and a secret key $sk \in SK$.*

$\texttt{SharedKey}(pk_1, sk_2) : PK \times SK \rightarrow S$
: *takes as input a public key $pk_1 \in PK$ and a secret key $sk_2 \in SK$ and outputs a shared secret $s \in S$.*

For correctness, it is required that, for any key pairs $(pk_1, sk_1) \in PK \times SK$ and $(pk_2, sk_2) \in PK \times SK$ generated by the algorithm `KeyGen` the algorithm `SharedKey` satisfies the constraint:

$$\texttt{SharedKey}(pk_1, sk_2) = \texttt{SharedKey}(pk_2, sk_1).$$

The most common two models for analyzing the security of PKEs and KEMs are the *indistinguishability under chosen-plaintext attacks* (IND-CPA) and the *indistinguishability under adaptive chosen-ciphertext attacks* (IND-CCA).

Definition 14 (IND-CPA)**.** *Let PKE be a public-key encryption scheme defined by the tuple of probabilistic algorithms (`KeyGen`, `Encrypt`, `Decrypt`) with message space M. Consider the IND-CPA security game as in Figure 2.2. For a PKE we define the advantage of $\mathcal{A}$ winning the game as*

$$\mathsf{Adv}_{PKE}^{\text{IND-CPA}}(\mathcal{A}) := \left|\Pr[\text{IND-CPA}(\mathcal{A}) = 1] - \frac{1}{2}\right|.$$

```
Game IND-CPA($\mathcal{A}$)
$(pk, sk) \leftarrow \texttt{KeyGen}()$
$b \leftarrow\!\!\$ \{0,1\}$
$(m_0, m_1) \leftarrow \mathcal{A}(pk)$
$c \leftarrow \texttt{Encrypt}(pk, m_b)$
$b' \leftarrow \mathcal{A}(pk, c)$
return $b = b'$
```

Figure 2.2: IND-CPA game for a PKE.

In the IND-CPA model (passive attacks) an adversary with knowledge of two candidates for the plaintext is not able to distinguish which one is related to the challenged ciphertext.

Definition 15 (IND-CCA)**.** *Let KEM be a key-encapsulation mechanism defined by the tuple of probabilistic algorithms (`KeyGen`, `Encaps`, `Decaps`) with message space M. Consider the IND-CCA security game as in Figure 2.3. For a KEM we define the advantage of $\mathcal{A}$ winning the game as*

$$\mathsf{Adv}_{KEM}^{\text{IND-CCA}}(\mathcal{A}) := \left|\Pr[\text{IND-CCA}(\mathcal{A}) = 1] - \frac{1}{2}\right|.$$

```
Game IND-CCA($\mathcal{A}$)
$(pk, sk) \leftarrow \texttt{KeyGen}()$
$b \leftarrow\!\!\$ \{0,1\}$
$(m_0, m_1) \leftarrow \mathcal{A}(pk)$
$(s_0, c^*) \leftarrow \texttt{Encaps}(pk)$
$s_1 \leftarrow\!\!\$ S$
$b' \leftarrow \mathcal{A}^{\texttt{DEC}}(pk, c, s_b)$
return $b = b'$

Oracle DEC(c)
if $c \neq c^*$
    return $\texttt{Decaps}(sk, c)$
else
    return $\perp$
```

Figure 2.3: IND-CCA game for a KEM.

In the IND-CCA model (active attacks) the more powerful adversary is given access to an oracle which decrypts all chosen but the challenged ciphertexts. Again, the adversary should not be able to distinguish between two ciphertexts.

Fujisaki and Okamoto [88] introduced a transformation that transforms CPA-secure schemes into CCA-secure schemes. We remark that any secure PKE with sufficiently large plaintext space can be trivially turned into a secure KEM and vice versa. Further, according to [87] any secure NIKE [79] scheme can be generically converted into an IND-CCA secure PKE scheme.

2.3.2 Digital Signature Schemes

Digital signatures are commonly used for achieving integrity and authenticity of data.

Definition 16 (Digital Signature Schemes)**.** *Let M be a message space, PK be a public-key space, SK be a private-key space, and SM be a signed-message space, then a signature scheme is defined by the tuple (`KeyGen`, `Sign`, `Open`) as follows:*

`KeyGen`$() : \emptyset \rightarrow PK \times SK$
: *takes no inputs and outputs a key pair consisting of a public key $pk \in PK$ and a secret key $sk \in SK$.*

`Sign`$(sk, m) : SK \times M \rightarrow SM$
: *takes as input a secret key $sk \in SK$ and a message $m \in M$ and outputs a signed message $sm \in SM$.*

`Open`$(pk, sm) : PK \times SM \rightarrow M \cup \{\perp\}$
: *takes as input a public key $pk \in PK$ and a signed message $sm \in SM$ and outputs either the message $m \in M$ if the signature is valid or an error $\perp$ if the signature is not valid, respectively.*

We note that in the standard definition[9] of digital signature schemes `Sign`(sk, m) returns a signature rather than a signed message.

For correctness, it is required that, for any key pairs $(pk, sk) \in PK \times SK$ generated by the algorithm `KeyGen` and any message $m \in M$ the algorithms `Open` and `Sign` satisfy the constraint:

$$\texttt{Open}(pk, \texttt{Sign}(sk, m)) = m.$$

The most common model for analyzing the security of digital signature schemes is the *existential unforgeability under adaptive chosen-message attacks* (EU-CMA) [98, 99].

[9]Following the *Sign/Verify* definition instead of the *Sign/Open* definition required by NIST.

Definition 17 (EU-CMA)**.** *Let SIG be a signature scheme defined by the tuple (*`KeyGen`*,* `Sign`*,* `Open`*) with message space M. Consider the EU-CMA security game as in Figure 2.4. For a signature scheme we define the advantage of $\mathcal{A}$ winning the game as*

$$\mathsf{Adv}_{SIG}^{\text{EU-CMA}}(\mathcal{A}) := \Pr[\text{EU-CMA}(\mathcal{A}) = 1].$$

```
Game EU-CMA($\mathcal{A}$)
$(pk, sk) \leftarrow$ KeyGen()
$M_\mathcal{A} \leftarrow \emptyset$
while
  $m' \leftarrow \mathcal{A}(pk)$
  $sm' \leftarrow \mathcal{A}^{\texttt{SIGN}}(pk, m')$
  $M_\mathcal{A} = M_\mathcal{A} \cup m'$
endwhile
$m^* \leftarrow \mathcal{A}(pk)$
$sm^* \leftarrow \mathcal{A}^{\texttt{FORGE}}(pk, m^*)$
return [Open$(pk, sm^*) = m^*$] $\wedge$ [$m^* \notin M_\mathcal{A}$]
```

```
Oracle SIGN(m)
return Sign$(sk, m)$
```

Figure 2.4: EU-CMA game for a signature scheme.

In the EU-CMA model, an adversary with access to a signing oracle must not be able to produce a valid signature for any message, which has not been signed by the signing oracle.

2.3.3 Threshold Schemes

The main motivation for threshold cryptography is to provide techniques allowing to securely distribute trust in the operation of cryptographic primitives. Informally, a threshold scheme (or secret-sharing scheme) allows to share a secret among shareholders such that any collaborating set with a sufficient number of participants can recover the shared secret, while any other (unauthorized) set is not able to get any additional information about the shared secret. In the following, we present the definition for the common case of *k-out-of-n* and refer to Section 6.1 for more details.

Definition 18 ((k,n)-Secret Sharing Schemes)**.** *Let S be a key space, then a (k,n) or k-out-of-n secret-sharing scheme is defined by the pair (*`Share`*,* `Rec`*) as follows:*

`Share`$(s) : S \rightarrow S^n$

> *takes as input a key $s \in S$ and outputs a n-tuple of shares $(s_1, \ldots, s_n)$, where $s_1, \ldots, s_n \in S$.*

`Rec`$(s_1, \ldots, s_j) : S^j \to S$

takes as input a j-tuple of shares $(s_1, \ldots, s_j) \in S$ with $k \leq j \leq n$ and outputs the shared key $s \in S$.

For correctness, it is required that, for any key $s \in S$ and for any subset S' with $\#S' \geq k$ of `Share`(s), the algorithm `Rec` satisfies the constraint

$$\texttt{Rec}(S') = s.$$

There exists no algorithm `A` such that `A`$(S'') = s$ with $\#S'' < k$. Further, the distribution of `Share`(s) and `Share`(s') for any $s, s' \in S$ should be identical.

2.4 Cryptographic Protocols

In this section we briefly describe two key-exchange protocols based on isogenies. In particular, we will start with elliptic-curve Diffie–Hellman (ECDH) as a basis and then look at two isogeny-based protocols, which are the main focus of interest in this thesis.

The most fundamental basis for such public-key exchange protocols is the Diffie–Hellman key-agreement scheme [77]. First based on the discrete-logarithm problem on multiplicative groups of finite fields, it was adapted to elliptic-curve discrete logarithms by Miller [139] and Koblitz [115].

The simplified workflow of the elliptic-curve Diffie–Hellman (ECDH) protocol, as shown in Figure 2.5, is as follows. First, both participants (Alice and Bob) agree on a set of domain parameters: An elliptic curve E over a field $\mathbb{F}_p$ with a prime p and a point $P \in E$ of prime order ℓ generating the cyclic subgroup of the form $\langle P \rangle = \{\infty_E, P, [2]P, [3]P, ..., [(\ell - 1)]P\}$. Both parties randomly sample a private key a resp. b where $a, b \in (\mathbb{Z}/\ell\mathbb{Z})^*$ and calculate the public key $P_a = [a]P$ resp. $P_b = [b]P$ using the point P as a generator. Then Alice's key pair is (P_a, a) and Bob's key pair is (P_b, b). The parties exchange each other's public keys. Finally, the result calculated by both parties is equal, since $[b]P_a = [ba]P = [ab]P = [a]P_b$ due to commutativity of the group law, and thus a shared secret. The security of this key exchange protocol relies on the hardness of the following two computational problems.

Problem 1 (Computational Diffie–Hellman problem (CDH))**.** *Given P, $[a]P$, and $[b]P \in E(\mathbb{F}_p)$, compute $[ab]P$.*

Problem 2 (Decisional Diffie–Hellman problem (DDH))**.** *Given P, $[a]P$, and $[b]P \in E(\mathbb{F}_p)$, and a point $Q \in E(\mathbb{F}_p)$ determine whether or not $Q = [ab]P$.*

We note that DDH reduces to CDH [76].

Although these problems are believed to be hard in the classical setting, they can be solved efficiently using Shor's algorithm [170] on a sufficiently large quantum computer.

Instead of working with points on a single elliptic curve, isogeny-based cryptography is based on isogenies between elliptic curves, as shown in Figure 2.6. The next sections introduce two isogeny-based schemes which are supposed to be quantum resistant. All the results in the remainder of this thesis are based on these schemes.

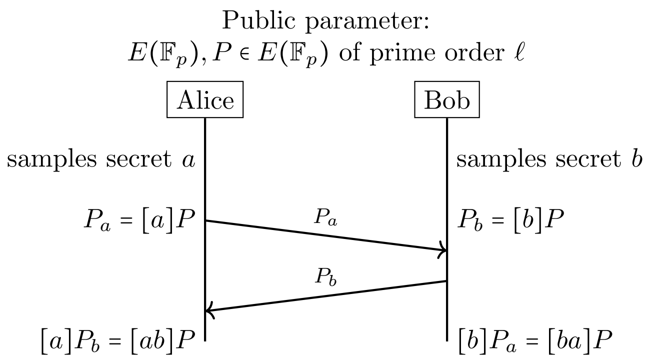


Figure 2.5: Elliptic-curve Diffie–Hellman (ECDH).

$$\begin{tikzcd}
E \arrow[r, "\varphi_A"] \arrow[d, "\varphi_B"'] & E_A \arrow[d, "\varphi'_B"] \\
E_B \arrow[r, "\varphi'_A"'] & E_{AB}
\end{tikzcd}$$

Figure 2.6: High-level view of an isogeny-based DH protocol where isogenies are denoted by φ_i and φ'_i where $i \in \{A, B\}$.

2.4.1 Supersingular Case over $\mathbb{F}_{p^2}$ (SIDH)

In [109], Jao and De Feo proposed a new approach for public-key cryptographic protocols based on the difficulty of computing isogenies between supersingular elliptic curves. The supersingular isogeny Diffie–Hellman (SIDH) protocol uses isomorphism classes of supersingular curves over $\mathbb{F}_{p^2}$. In the following, we review the most fundamental ideas and refer to [109] for more details on SIDH.

The simplified workflow of the SIDH protocol, as shown in Figure 2.7, is as follows. First, we choose a prime $p = \ell_A^{e_A} \cdot \ell_B^{e_B} - 1$ such that ℓ_A, ℓ_B are small distinct primes and $\ell_A^{e_A} \approx \ell_B^{e_B}$. Further, we work with a curve E such that $\#E(\mathbb{F}_{p^2}) = (p+1)^2$ which means that all points of order $\ell_A^{e_A}$ and all points of order $\ell_B^{e_B}$ (the torsion groups $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively) are defined over $\mathbb{F}_{p^2}$. Moreover, we can use points P_A and Q_A of order $\ell_A^{e_A}$ as basis for the subgroups

$$\langle P_A \rangle, \langle P_A + Q_A \rangle, \langle P_A + [2]Q_A \rangle, \dots, \langle P_A + [\ell_A^{e_A} - 1]Q_A \rangle,$$

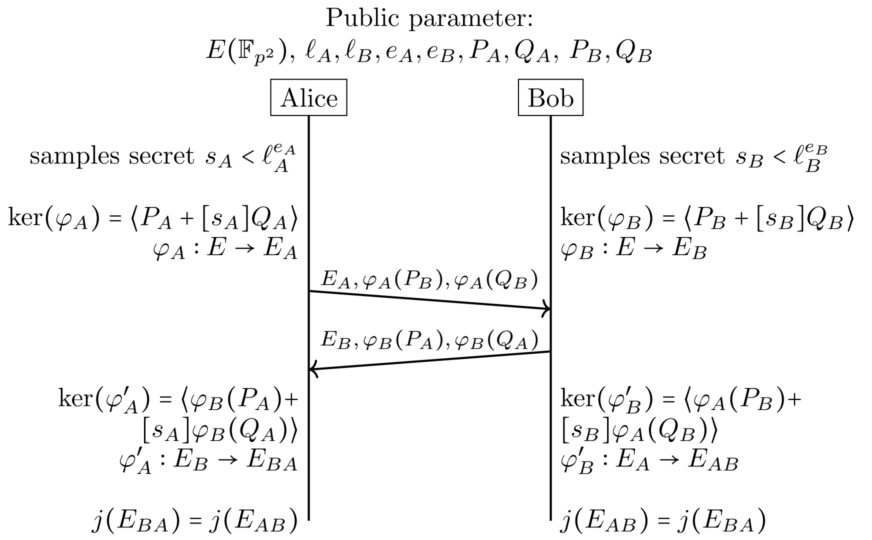

Public parameter:
$E(\mathbb{F}_{p^2})$, $\ell_A, \ell_B, e_A, e_B, P_A, Q_A$, P_B, Q_B
Alice
Bob
samples secret $s_A < \ell_A^{e_A}$
samples secret $s_B < \ell_B^{e_B}$
$\ker(\varphi_A) = \langle P_A + [s_A]Q_A \rangle$
$\varphi_A : E \to E_A$
$\ker(\varphi_B) = \langle P_B + [s_B]Q_B \rangle$
$\varphi_B : E \to E_B$
$E_A, \varphi_A(P_B), \varphi_A(Q_B)$
$E_B, \varphi_B(P_A), \varphi_B(Q_A)$
$\ker(\varphi'_A) = \langle \varphi_B(P_A) + [s_A]\varphi_B(Q_A) \rangle$
$\varphi'_A : E_B \to E_{BA}$
$\ker(\varphi'_B) = \langle \varphi_A(P_B) + [s_B]\varphi_A(Q_B) \rangle$
$\varphi'_B : E_A \to E_{AB}$
$j(E_{BA}) = j(E_{AB})$
$j(E_{AB}) = j(E_{BA})$

Figure 2.7: SIDH key exchange.

where each of these subgroups determines a unique isogeny (up to isomorphism) of order $\ell_A^{e_A}$. This applies analogously to points P_B and Q_B of order $\ell_B^{e_B}$. Thus, the set of domain parameters are $p, E, \ell_A, \ell_B, e_A, e_B$, and a basis P_A, Q_A of the $\ell_A^{e_A}$-torsion on $E(\mathbb{F}_{p^2})$, and similarly a basis P_B, Q_B of the $\ell_B^{e_B}$-torsion on $E(\mathbb{F}_{p^2})$ as described above. Both participants randomly sample a private key $s_i \in \{0, 1, \ldots, \ell_i^{e_i} - 1\}$, calculate $P_i + [s_i]Q_i$ and the secret isogeny φ_i with kernel[10] $\ker(\varphi_i) = \langle P_i + [s_i]Q_i \rangle$ where $i \in \{A, B\}$. In order to agree on a shared secret, both parties have to calculate an isogeny φ'_A on E_B and φ'_B on E_A that are analogous to φ_A resp. φ_B. Since there is no corresponding torsion basis points on E_A, E_B publicly available, both parties have to include additional points in their public keys. For this, Alice additionally sends the image of Bob's torsion basis points $\varphi_A(P_B), \varphi_A(Q_B)$ and vice versa. In summary, Alice's secret key is s_A and her public key is $(E_A, \varphi_A(P_B), \varphi_A(Q_B))$. This applies analogously to Bob. Since the resulting curves E_{BA} and E_{AB} are isomorphic, their j-invariant can serve as a shared secret. Figure 2.8 shows a toy example for the isogeny class of the supersingular curve $E : y^2 = x^3 + x$ over $\mathbb{F}_{431^2}$ where $p = 2^4 \cdot 3^3 - 1 = 431$.

Computing the full isogeny φ_i at once would be very inefficient because the cost grows linearly with the degree. Hence, SIDH decomposes this $\ell_i^{e_i}$ isogeny into e_i computations of ℓ_i-isogenies. For optimal strategies on calculating such isogenies, we refer to [109].

[10]Instead of using kernels of the form $\langle [s'_i]P_i + [s_i]Q_i \rangle$ with almost no loss in generality, one can use private keys with $s'_i = 1$ [63].

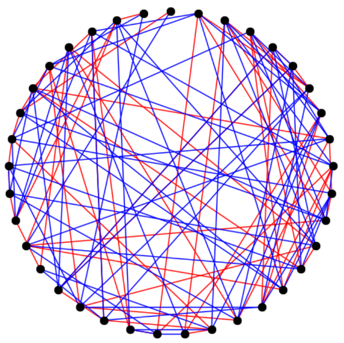


Figure 2.8: Graph with isomorphism classes of supersingular curves over $\mathbb{F}_{431^2}$ with 2-isogenies (red) and 3-isogenies (blue).

The security of the SIDH key-exchange protocol relies on the hardness of the following problems.

Problem 3 (Supersingular Decision Diffie-Hellman (SSDDH) Problem)**.** *Given the public parameters $\ell_A, \ell_B, e_A, e_B, p, E, P_A, Q_A, P_B, Q_B$ the elliptic curves E_A, E_B, the points $\phi_A(P_B), \phi_A(Q_B), \phi_B(P_A), \phi_B(Q_A)$, and a tuple sampled with probability $1/2$ from one of the two distributions:*

- *$(E_A, E_B, \phi_A(P_B), \phi_A(Q_B), \phi_B(P_A), \phi_B(Q_A), E_{AB})$ with*

$$E_{AB} \cong E/\langle [m_A]P_A + [n_A]Q_A, [m_B]P_B + [n_B]Q_B \rangle.$$

- *$(E_A, E_B, \phi_A(P_B), \phi_A(Q_B), \phi_B(P_A), \phi_B(Q_A), E_C)$ with*

$$E_C \cong E/\langle [m'_A]P_A + [n'_A]Q_A, [m'_B]P_B + [n'_B]Q_B \rangle$$

  where m'_A, n'_A, m'_B, n'_B are chosen randomly. Determine from which distribution the tuple is sampled.

Problem 4 (Supersingular Computational Diffie–Hellman (SSCDH) Problem)**.** *Given the public parameters $\ell_A, \ell_B, e_A, e_B, p, E, P_A, Q_A, P_B, Q_B$ the elliptic curves E_A, E_B, and the points $\phi_A(P_B), \phi_A(Q_B), \phi_B(P_A), \phi_B(Q_A)$, find the j-invariant of $E/\langle P_A + [s_A]Q_A, P_B + [s_B]Q_B \rangle$.*

Analogously to DDH and CDH, we note that SSDDH reduces to SSCDH [182]. Further, both the decisional and computational assumptions depend on the following computational problem.

Problem 5 (Computational Supersingular Isogeny (CSSI) Problem)**.** *Given the public parameters $\ell_A, \ell_B, e_A, e_B, p, E, P_A, Q_A, \phi_A(P_B), \phi_A(Q_B)$ and the elliptic curve E_A, compute a degree-$\ell_A^{e_A}$ isogeny $\varphi_A : E \to E_A$.*

The fact of having additional points $\phi_A(P_B), \phi_A(Q_B)$ as part of the public keys raised some concern due to possible so-called *torsion-point attacks* [86, 75]. Regarding the security of the underlying problems on the classical side, since the key spaces have size $\approx \sqrt{p}$, generic meet-in-the-middle attacks have a complexity of $O(\sqrt[4]{p})$. However, based on [1] the effective costs of such attacks led to a reduction of the first proposed parameter sizes. On the quantum side, the initially proposed parameters should achieve a higher security level based on the effective costs of the quantum computation (RAM model) of a claw finding attack [175] with complexity of $O(\sqrt[6]{p})$ [111].

SIDH is the basis of a KEM called *supersingular isogeny key encapsulation* (SIKE) [108], which is the only isogeny-based submission to the NIST PQC process and currently under consideration by the NIST as a 4th round candidate.

Remark 2. *Because of the attacks presented in [46, 132], the SIKE team states that SIKE and SIDH should not be used.*[11]

2.4.2 Supersingular Case over $\mathbb{F}_p$ (CSIDH)

In 2018, Castryck, Lange, Martindale, Panny, and Renes published the *commutative supersingular isogeny Diffie–Hellman* (CSIDH) scheme [49]. The presented key-exchange scheme is based on the works of Couveignes [66] and Rostovtsev–Stolbunov [164]. While the scheme of Couveignes and Rostovtsev–Stolbunov is based on isogenies between ordinary elliptic curves, CSIDH works with isogenies between isomorphism classes of supersingular curves defined over $\mathbb{F}_p$ where p is prime. In this section, we only present the algorithmic aspects of CSIDH and refer to [49] for the mathematical background and a more detailed description.

CSIDH works on a set of curves with $\#E(\mathbb{F}_p) = p + 1$ and the same $\mathbb{F}_p$-rational endomorphism ring. Thereby, isogenies are represented by ideal classes in this ring forming a group. Thus, an ideal $\mathfrak{a}$ can act on a curve E to produce the codomain curve E_A, denoted as the CSIDH group action $\mathfrak{a} * E = E_A$. The CSIDH protocol chooses p such that $p + 1 = 4\prod_{i=1}^{n} \ell_i$ where $\ell_1, \ldots, \ell_n$ are distinct odd primes.

In particular, we are interested in specific isogenies $\mathfrak{l}_i$ (of degree ℓ_i) defined by the kernel $G = E[\ell_i] \cap E[\pi - 1]$, where π stands for the Frobenius endomorphism (see Definition 7), i.e., $\mathbb{F}_p$-rational points that have ℓ_i-torsion. As E has $p + 1$ points over $\mathbb{F}_p$, we get

$$E(\mathbb{F}_p) \cong \mathbb{Z}/(p+1) \cong \mathbb{Z}/4 \times \mathbb{Z}/\ell_1 \times \cdots \times \mathbb{Z}/\ell_n.$$

This implies that for each ℓ_i there exist non-trivial $\mathbb{F}_p$-rational points $P \in E[\ell_i] \cap E[\pi - 1]$, and that all but one of them, ∞_E, will generate the kernel

[11] `https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/round-4/submissions/sike-team-note-insecure.pdf`

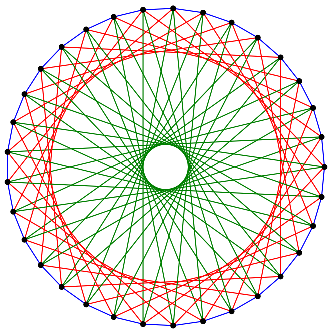


Figure 2.9: ℓ_i-isogeny graph for degrees 3 (blue), 5 (green), and 11 (red) (*Union of cycles*).

G. Since the codomain curve E' of such an isogeny is again supersingular and so $\#E'(\mathbb{F}_p) = p+1$ (see Theorem 4), $\mathfrak{l}_i$ must also be applicable to E'. This implies a group action of the elements $\mathfrak{l}_i$ on the supersingular curves E over $\mathbb{F}_p$, which we denote by $[\mathfrak{l}_i] * E$. This group action is commutative:

$$[\mathfrak{l}_i\mathfrak{l}_j] * E \cong [\mathfrak{l}_i] * ([\mathfrak{l}_j] * E) \cong [\mathfrak{l}_j] * ([\mathfrak{l}_i] * E) \cong [\mathfrak{l}_j\mathfrak{l}_i] * E.$$

The dual isogeny (see Theorem 6) associated to $\mathfrak{l}$ is denoted as $\mathfrak{l}^{-1}$, and is defined by the kernel $G = E[\ell_i] \cap E[\pi + 1]$.

Thus, due to the commutativity of the underlying group action, the NIKE based on CSIDH is very similar to ECDH, as shown in Figure 2.10. Both parties first sample a secret $\mathfrak{a}$ resp. $\mathfrak{b}$ by sampling an exponent vector $(a_1, \ldots, a_n)$ with $|a_i| \leq M$ for a given bound $M \in \mathbb{N}$. This list of exponents represents an ideal class $\mathfrak{a} = \prod_{i=1}^{n} \mathfrak{l}_i^{a_i}$ such that

$$E_A = \mathfrak{a} * E = \mathfrak{l}_1^{a_1} * (\cdots * (\mathfrak{l}_n^{a_n} * E)).$$

In fact, for calculating the action of the ideals $\mathfrak{l}_i^{a_i}$ for $i = 1, \ldots, n$, we compute $|a_i|$ ℓ_i-isogenies. After computing the group action, as described in Algorithm 1, they exchange each other's public keys $E_A = \mathfrak{a} * E$ resp. $E_B = \mathfrak{b} * E$. Finally, the result calculated by both parties is equal since $E_{BA} = \mathfrak{a} * E_B = \mathfrak{ab} * E = \mathfrak{b} * E_A = E_{AB}$ and thus represents a shared secret. Figure 2.9 shows a toy example for the isogeny class of the supersingular curve $E : y^2 = x^3 + x$ over $\mathbb{F}_{659}$ where $p = 659 = 4 * 3 * 5 * 11 - 1$.

In CSIDH, no further information such as evaluated torsion points as in SIDH are included in public keys. Thus, the security of CSIDH relies on the hardness of the following pure isogeny-finding problems.

Problem 6 (Group Action Inverse Problem (GAIP)). *Given two curves E, E_{AB}, find the ideal class $\mathfrak{a}$ such that $E_{AB} = \mathfrak{a} * E$.*

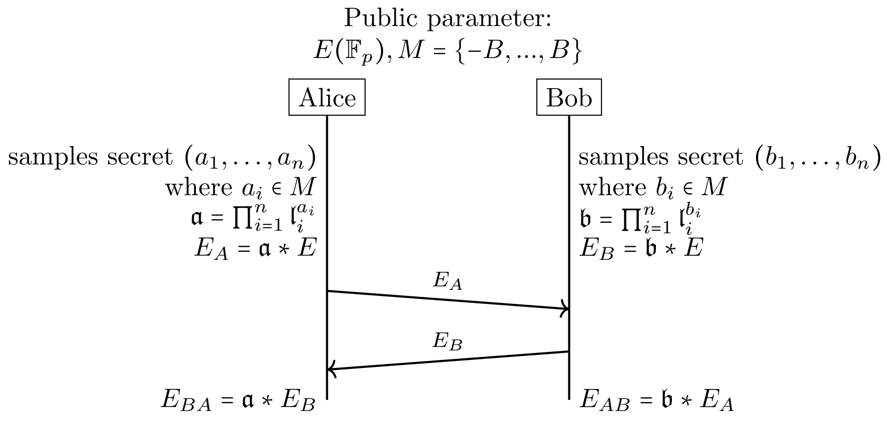

Public parameter:
$E(\mathbb{F}_p), M = \{-B, ..., B\}$
Alice
Bob
samples secret $(a_1, \dots, a_n)$
where $a_i \in M$
$\mathfrak{a} = \prod_{i=1}^{n} \mathfrak{l}_i^{a_i}$
$E_A = \mathfrak{a} * E$
samples secret $(b_1, \dots, b_n)$
where $b_i \in M$
$\mathfrak{b} = \prod_{i=1}^{n} \mathfrak{l}_i^{b_i}$
$E_B = \mathfrak{b} * E$
E_A
E_B
$E_{BA} = \mathfrak{a} * E_B$
$E_{AB} = \mathfrak{b} * E_A$

Figure 2.10: CSIDH key exchange.

Assuming that an adversary is able to compute the shared secret E_{AB} without being able to solve the GAIP problem, we define the following problem.

Problem 7 (Group Action Computational Diffie–Hellman Problem (GAIP-CDH))**.** *Given $E, E_A = \mathfrak{a} * E$, and $E_B = \mathfrak{b} * E$, find E_{AB} such that $E_{AB} = \mathfrak{a} * (\mathfrak{b} * E)$.*

The GAIP-CDH is often called *parallelisation problem.* The intuitive decisional version of this problem is as follows.

Problem 8 (Group Action Decisional Diffie–Hellman Problem (GAIP-DDH))**.** *Given $E_A = \mathfrak{a} * E, E_B = \mathfrak{b} * E$ and $E_C = \mathfrak{c} * E$, distinguish whether $\mathfrak{c} = \mathfrak{a} \cdot \mathfrak{b}$ or $\mathfrak{c}$ was randomly sampled.*

From a classical perspective, the security of CSIDH is related to the size of the key space. Thus, the key space has to be large enough to protect against brute force attacks or meet-in-the-middle variants. On the quantum side, some recent works [35, 158, 54] recommend increasing the underlying prime sizes within CSIDH. However, the exact quantum security of CSIDH is still subject of debate. For further details on the quantum security of CSIDH, we refer to [54].

Algorithm 1: Evaluating the class group action.

```
Input  : $a' \in \mathbb{F}_p$ such that $E : y^2 = x^3 + a'x^2 + x$ is supersingular, and
         a list of integers $(a_1, ..., a_n)$ with $a_i \in M$ for all $i \leq n$.
Output: $a \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_A$.
1  while some $a_i \neq 0$ do
2      Sample a random $x \in \mathbb{F}_p$.
3      Set $s \leftarrow +1$ if $x^3 + a'x^2 + x$ is a square in $\mathbb{F}_p$, else $s \leftarrow -1$.
4      Let $S = \{i \mid sign(a_i) = s\}$.
5      if $S = \varnothing$ then
6          Go to line 2.
7      $P = (x : 1)$, $k \leftarrow \prod_{i \in S} \ell_i$, $P \leftarrow [(p+1)/k]P$.   // $p = 4 \cdot \ell_1 \cdot ... \cdot \ell_n - 1$
8      foreach $i \in S$ do
9          $K \leftarrow [k/\ell_i]P$. if $K \neq \infty$ then
10             Compute a degree-$\ell_i$ isogeny $\varphi : E \to E_A$ with
                $ker(\varphi) = \langle K \rangle$.
11             $a' \leftarrow a$, $P \leftarrow \varphi(P)$, $k \leftarrow k/\ell_i$, $a_i \leftarrow a_i - s$.
12 return $a$.
```

2.5 Physical Attacks

In the following sections, we introduce the basic aspects of physical attacks and refer to [133, 183] for further details on this topic.

Cryptanalysis refers to the process of analyzing cryptographic systems looking for vulnerabilities or leaks of secret information. In classical or mathematical cryptanalysis, where a black-box model is applied, the adversaries are only allowed to interact with the cryptographic primitive over the predefined channels. Thus, only plaintexts and ciphertexts are available for analysis (see models for analyzing security in Section 2.3). However, once cryptographic primitives are deployed on physical devices, the assumption of the black-box model is missing an addition channel of information. Therefore, in practical cryptanalysis, to reveal secret information, the adversaries are able to observe and measure physical and exploitable information leaked from a device and/or apply fault attacks by interacting with a target device. Thus, regardless of the assumed security of the underlying cryptographic primitive against classical cryptanalysis, implementation of cryptographic algorithm deployed on a specific physical device may be weak against physical attacks. The trade-off between performance and security is a key factor when designing countermeasures against such attacks, e.g. [186].

In general, there are two well established types of physical attacks: side-channel attacks (SCA) and fault attacks (FA), which can further, depending on how invasive they are, be divided into invasive, semi-invasive, and non-invasive categories. The main difference between SCA and FA is whether any interaction with the target device is required in the attack setup. While SCA extract information by exploiting physical measurements produced during execution, in FA, the adversaries actively tamper with the device by inducing faults during the cryptographic computation.

Besides SCA and FA, attacks combining SCA and FA, called *hybrid attacks*, have been proposed [96, 127]. In this kind of attack, the adversaries are able to gain specific information leakage that only appears in a fault-injected environment.

2.5.1 Side-Channel Attacks

First introduced in 1996 by Kocher [118], SCAs aim to extract secret information through measurement and analysis of physical information during the execution of a cryptographic algorithm. Informally, this is achieved by identifying correlation between secret information and runtime behaviors. Such correlations are typically based on the observation of power consumption [185], time consumption [118], or electromagnetic radiation [81]. Many approaches have been proposed to exploit such side-channel information, including simple side-channel analysis (SSCA) [118], differential side-channel analysis (DSCA) [118] and template attacks (TA) [52].

In this thesis, we focus on SSCA and DSCA. In particular, we apply correlation-collision SCA [142] on isogeny-based schemes. In such a correlation-collision SCA, the attacker concludes from the leakage that identical (value collision) or very similar (strongly related) intermediate values have been processed and based on that, deduces secret information. Among other models for estimating such a correlation, the Hamming weight and the Hamming distance model are the most widely used ones.

We refer to Section 5.3, where we apply correlation-collision SCA based on the Hamming weight model to fully recover secret keys of SIKE and multiple variants of CSIDH. We further refer to [130, 11, 125] for more details on side-channel attacks.

2.5.2 Fault Attacks

First proposed in [33], FAs provide a powerful approach to recover secret information of cryptographic primitives on physical devices. In FAs, the adversary is assumed to have physical access to the target device during execution. The main idea behind FAs is to disturb the execution of a running algorithm to obtain faulty outputs or faulty behavior. On a physical device, a fault injected can, e.g., skip instructions, lead to an unpredictable memory state, access non-permitted location of memory, or perform a non-permitted operation. Numerous fault-injection techniques (e.g., clock glitches, power glitches, laser shots, and electromagnetic interference) have been proposed and exploited to attack cryptographic schemes [37].

Among other approaches for fault attacks, safe-error attacks, introduced in 2000 by Yen and Joye [188], provide a powerful technique to exploit fault injection. In this kind of attacks, after injecting a fault, the output of the computation reveals secret information based on the executed path within the attacked algorithm. There are two types of safe-error attacks: the computational (C safe-error), targeting algorithmic vulnerabilities [189] and the memory (M safe-error), focussing on vulnerabilities within specific implementations [112].

We refer to Section 5.1, where we evaluate the impact of C safe-error attacks on constant-time implementations of CSIDH. Further, we refer to Section 5.2, where we present C and M safe-error attacks against SIKE and a constant-time implementation of CSIDH.

For more details on fault attacks, we refer to [13, 11].

3 Optimizations

3.1 On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic

This chapter is for all practical purposes identical to the paper *On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic* [138] authored jointly with Michael Meyer and Steffen Reith.

3.1.1 Introduction

In the recent years, the topic of post-quantum cryptography (PQC) has gained a massive boost of attention and research. The threat of a possible feasibility of building large quantum computers, that could break, e.g., elliptic curve cryptography like ECDH by Shor's algorithm [171], led to a competition by NIST for the standardization of PQC primitives [178].

One proposal for a PQC key establishment protocol is based on isogenies between elliptic curves. First proposed by Couveignes in 1997 [66], followed by Rostovtsev and Stolbunov in 2006 [164], using ordinary elliptic curves, Jao and De Feo proposed the use of supersingular elliptic curves in 2011, in order to obtain a quantum-resistant scheme [109]. Since 2016, when Costello, Longa and Naehrig published an efficient algorithm for the isogeny-based key exchange [63], SIDH has gained a lot of attention and research. The use of twisted Edwards curves has been suggested by Costello and Hisil in [62].

In the mentioned NIST standardization competition [178], there is a submission based on SIDH. SIKE (see [108]) is a key encapsulation mechanism, that is built upon the key exchange protocol, that we describe here. In addition, there are signature schemes (see [191], [92]) and authenticated key exchange protocols (see [90], [129], [122]) based on SIDH. Therefore, they all inherit the following performance discussions.

3.1.2 Preliminaries

Isogenies are defined as non-constant morphisms between elliptic curves, that preserve the identity elements. Separable isogenies are uniquely determined by their kernel, and the degree of the isogeny is the cardinality of its kernel.

In the SIDH key-exchange primitive, isogenies of large degree are computed as a composition of small-degree isogenies. We work over a field $\mathbb{F}_{p^2}$ with a prime of the form $2^m 3^n f \pm 1$ with a small integer f. We choose an initial supersingular elliptic curve E_0 over $\mathbb{F}_{p^2}$ and want to compute a 2^m- and 3^n-isogeny respectively for each party, in the following called Alice and Bob. Therefore for generating the kernels of the isogenies, whose sizes determine the degrees of the isogenies, we choose initial points P_A, Q_A, P_B, Q_B lying in the corresponding torsion groups, namely $P_A, Q_A \in E_0[2^m]$ and $P_B, Q_B \in E_0[3^n]$.
In the following, we describe Alice's key generation, while Bob's key generation works in an analogous way.

Alice chooses a random integer m_A and computes $R_0 = P_A + [m_A]Q_A$ as generator of the kernel for computing her 2^m-isogeny. However, since computing large-degree isogenies is expensive, m isogenies of degree 2 are computed. Therefore, we compute $[2^{m-1}]R_0$ as generator of the kernel of the 2-isogeny φ_0, that maps to $E_1 = E_0/\langle[2^{m-1}]R_0\rangle$. φ_0 can be computed by Vélu's formulae [184]. In addition, $R_1 = \varphi_0(R_0)$ is computed. The next step is to compute $[2^{m-2}]R_1$, φ_1, $E_2 = E_1/\langle[2^{m-2}]R_1\rangle$, and $R_2 = \varphi_1(R_1)$. Following this pattern, we obtain a 2^m-isogeny φ_A as composition of m isogenies of degree 2, that maps from E_0 to a curve E_A.
The public key then consists of E_A (in terms of the curve parameters), $\varphi_A(P_B)$, and $\varphi_A(Q_B)$. Alice receives Bob's computed public key, which contains the curve E_B, $\varphi_B(P_A)$, and $\varphi_B(Q_A)$.
Following the same strategy again, Alice then computes $R_{BA} = \varphi_B(P_A) + [m_A]\varphi_B(Q_A)$ and uses this point as generator for computing the 2^m-isogeny φ_{BA}, again as a composition of 2-isogenies, that maps from E_B to E_{BA}. Similarly, Bob obtains a 3^n-isogeny φ_{AB}, that maps from E_A to E_{AB}. The shared secret can then be computed as j-invariant of the resulting curves, since the j-invariants of E_{AB} and E_{BA} are equal.

Instead of a field $\mathbb{F}_{p^2}$ with a prime of the form $2^m 3^n f \pm 1$, any other prime of the form $\ell_a^m \ell_b^n f \pm 1$ with ℓ_a, ℓ_b coprime can be used. However, the above mentioned choice seems to be the most efficient for SIDH. In [63] 4-isogenies are used instead of 2-isogenies for reasons of efficiency.

Furthermore, we note that there are better strategies to obtain isogenies of degree ℓ^m than the described multiplication-based approach. See [109] for a detailed analysis of optimal strategies.

3.1.3 Montgomery curve arithmetic

In their implementation of SIDH in [63], available at [65], Costello, Longa and Naehrig use elliptic curves in Montgomery form. They are given by an equation over a field $\mathbb{K}$ of the form

$$E_{a,b} : by^2 = x^3 + ax^2 + x.$$

To avoid inversions during point additions and doublings, projective coordinates can be used. Instead of the affine coordinates $(x, y) \in E_{a,b}$, we use $(X : Y : Z) \in \mathbb{P}^2$ with $x = X/Z$ and $y = Y/Z$, and $\mathcal{O}_E = (0 : 1 : 0)$ as point at infinity. If we embed this into $\mathbb{P}^1$ by dropping the Y-coordinate, we can use the efficient arithmetic given by Montgomery in [140]. Given a point $P_n = (X_n : Z_n)$, we can compute $[2]P_n = (X_{2n} : Z_{2n})$ by

$$\begin{aligned}
4X_nZ_n &= (X_n + Z_n)^2 - (X_n - Z_n)^2, \\
X_{2n} &= (X_n + Z_n)^2(X_n - Z_n)^2, \\
Z_{2n} &= (4X_nZ_n)((X_n - Z_n)^2 + ((A + 2)/4)(4X_nZ_n)).
\end{aligned}$$

Given another point $P_m = (X_m : Z_m)$ and the difference $P_{m-n} = P_m - P_n = (X_{m-n} : Z_{m-n})$, we can compute the sum $P_{m+n} = P_m + P_n = (X_{m+n} : Z_{m+n})$ by

$$\begin{aligned}
X_{m+n} &= Z_{m-n}((X_m - Z_m)(X_n + Z_n) + (X_m + Z_m)(X_n - Z_n))^2, \\
Z_{m+n} &= X_{m-n}((X_m - Z_m)(X_n + Z_n) - (X_m + Z_m)(X_n - Z_n))^2.
\end{aligned}$$

Thus a differential addition can be done using $4\mathbf{M} + 2\mathbf{S}$, or $3\mathbf{M} + 2\mathbf{S}$ if P_{m-n} is normalized, while a doubling needs $2\mathbf{M} + 2\mathbf{S} + 1\mathbf{C}$. As usual, we denote field multiplications by $\mathbf{M}$, field squarings by $\mathbf{S}$, and multiplications by a constant using $\mathbf{C}$. In our representation of computational costs, we ignore field additions and subtractions, since their costs are negligible in comparison to multiplications and squarings. We note that the formulae above lose information, since we do not distinguish between the possible corresponding coordinates Y and $-Y$. However, in the case of SIDH we do not need this information and it suffices to work entirely with $(X : Z) \in \mathbb{P}^1$.

In the SIDH implementation in [63], not only the point coordinates, but also the curve parameters are projectivized. Instead of a Montgomery curve of the form given above, we work with an equation of the form

$$E_{(A:B:C)} : By^2 = Cx^3 + Ax^2 + Cx,$$

where $(A : B : C) \in \mathbb{P}^2(K)$, such that $a = A/C$ and $b = B/C$ for the corresponding curve $E_{a,b}$. The j-invariants of the curves are then given by

$$j(E_{a,b}) = \frac{256(a^2-3)^3}{a^2-4} \quad \text{and} \quad j(E_{(A:B:C)}) = \frac{256(A^2-3C^2)^3}{C^4(A^2-4C^2)}.$$

From these formulae we see that the j-invariant does not depend on b or B, respectively. Therefore, it suffices to work with $(A : C) \in \mathbb{P}^1(K)$ in the projective model. Furthermore, neither the Montgomery arithmetic given above, nor the isogeny computations during SIDH require the coefficients b or B, respectively. However, we note that formulae that make use of the parameter a, like the Montgomery doubling above, must be trivially modified by substituting $a = A/C$ in order to work on $E_{(A:B:C)}$, as described in [63]. In this case a doubling costs $2\mathbf{M} + 2\mathbf{S} + 2\mathbf{C}$.

3.1.4 Twisted Edwards curve arithmetic

When it comes to elliptic-curve computations, Edwards curves are often the model of choice for fast arithmetic. However, in the context of SIDH, we have to compare the Edwards arithmetic formulae with the efficient XZ-only Montgomery arithmetic. On the other hand, most of the discussions about fast Edwards arithmetic focus on full-coordinate models.

Twisted Edwards curves over $\mathbb{K}$ are given by equations of the form

$$E_{E,a,d} : aX^2 + Y^2 = 1 + dX^2Y^2,$$

with $a, d \neq 0$, $d \neq 1$, and $a \neq d$. For $a = 1$, the twisted Edwards curve $E_{E,1,d} = E_{E,d}$ is called Edwards curve. As seen in the Montgomery case, projective coordinates can be used in order to avoid inversions during additions and doublings. However, in the Edwards case there are three more models, as described in [18].

Similarly to the Montgomery case above, we can use projective coordinates $(X : Y : Z) \in \mathbb{P}^2$ with $x = X/Z$ and $y = Y/Z$ for the corresponding affine point (x, y) on $E_{a,d}$. The projective curve equation is given by $aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2$. A projective point $P = (X_1 : Y_1 : Z_1)$ can be doubled using

$$\begin{aligned} &B = (X_1 + Y_1)^2; \quad C = X_1^2; \quad D = Y_1^2; \quad E = aC; \\ &F = E + D; \quad H = Z_1^2; \quad J = F - 2H; \\ &X_3 = (B - C - D) \cdot J; \quad Y_3 = F \cdot (E - D); \quad Z_3 = F \cdot J, \end{aligned}$$

where $[2]P = (X_3 : Y_3 : Z_3)$. Therefore, a doubling needs $3\mathbf{M} + 4\mathbf{S} + 1\mathbf{C}$ (see [17]).

Another model for twisted Edwards curves is given by the extended curve equation $aX^2 + Y^2 = Z^2 + dT^2$ with $XY = ZT$. Points on this curve are represented by $(X : Y : Z : T) \in \mathbb{P}^3$, where the corresponding affine point (x, y) is represented by $(x : y : 1 : xy)$. According to [103], there is a fast way to add two points $(X_1 : Y_1 : Z_1 : T_1)$ and $(X_2 : Y_2 : Z_2 : T_2)$ by

$$\begin{aligned}
&A = X_1 \cdot X_2;\;\; B = Y_1 \cdot Y_2;\;\; C = Z_1 \cdot T_2;\;\; D = T_1 \cdot Z_2;\; E = D + C;\\
&F = (X_1 - Y_1) \cdot (X_2 + Y_2) + B - A;\;\; G = B + aA;\;\; H = D - C;\\
&X_3 = E \cdot F;\;\; Y_3 = G \cdot H;\;\; Z_3 = F \cdot G;\;\; T_3 = E \cdot H,
\end{aligned}$$

where $(X_1 : Y_1 : Z_1 : T_1) + (X_2 : Y_2 : Z_2 : T_2) = (X_3 : Y_3 : Z_3 : T_3)$. The cost of this computation is $9\mathbf{M} + 1\mathbf{C}$.

Two more models are given by the inverted curve $aY^2Z^2 + X^2Z^2 = X^2Y^2 + dZ^4$ with points $(X : Y : Z) \in \mathbb{P}^2$ and the completed curve $aX^2T^2 + Y^2Z^2 = Z^2T^2 + dX^2Y^2$ with points $((X : Z), (Y : T)) \in \mathbb{P}^1 \times \mathbb{P}^1$ (see [18]). However, for these models there are no known doubling or addition formulae, that are more efficient than the ones mentioned above.

Similar to the XZ-only Montgomery curve arithmetic, Castryck, Galbraith, and Farashahi introduced a YZ-only doubling formula for Edwards curves in [48]. In the same way as presented in the mentioned paper, we can derive such a formula for twisted Edwards curves. For an affine point $P = (x, y)$, the twisted Edwards doubling formula is given by

$$[2]P = (x_2, y_2) = \left(\frac{2xy}{1 + dx^2y^2}, \frac{y^2 - ax^2}{1 - dx^2y^2} \right).$$

For the y-coordinate we have

$$\frac{y^2 - ax^2}{1 - dx^2y^2} = \frac{y^2(a - dy^2) - a(1 - y^2)}{(a - dy^2) - dy^2(1 - y^2)} = \frac{-dy^4 + 2ay^2 - a}{dy^4 - 2dy^2 + a},$$

where we make use of the curve equation $ax^2 + y^2 = 1 + dx^2y^2$. Expressing this in projective coordinates with $P = (X : Y : Z)$ and $[2]P = (X_2 : Y_2 : Z_2)$, we obtain

$$\begin{aligned}
Y_2 &= -dY^4 + 2aY^2Z^2 - aZ^4,\\
Z_2 &= dY^4 - 2dY^2Z^2 + aZ^4.
\end{aligned}$$

This can be computed using $5\mathbf{S} + 4\mathbf{C}$. Equivalent doubling formulae are also given in [134] with a cost of $5\mathbf{S} + 2\mathbf{C}$, since they use a single precomputed coefficient $\alpha = a/d$. In the case of an Edwards curve we have $a = 1$, so the cost decreases to $5\mathbf{S} + 2\mathbf{C}$. In [48] it is also pointed out that if ad is a square and $\sqrt{ad}$ is known, a doubling can be modified to cost $1\mathbf{M} + 3\mathbf{S} + 6\mathbf{C}$.

Similarly, in the Edwards case, if d is a square and $\sqrt{d}$ is known, a doubling takes $1\mathbf{M} + 3\mathbf{S} + 3\mathbf{C}$.

A formula for YZ-only differential addition of twisted Edwards curve points of odd order is derived in [134]. Given projective points $[n]P = (Y_n : Z_n)$ and $[n+1]P = (Y_{n+1} : Z_{n+1})$, and the affine y-coordinate of $P = (x, y)$, the point $[2n+1]P = (Y_{2n+1} : Z_{2n+1})$ can be computed with $4\mathbf{M} + 3\mathbf{S}$. If P is given projectively as $P = (Y : Z)$, we can modify the formulae to

$$\begin{aligned} Y_{2n+1} &= -Y((Y_n Z_{n+1})^2 + (Y_{n+1} Z_n)^2) + 2ZY_n Z_n Y_{n+1} Z_{n+1}, \\ Z_{2n+1} &= Z((Y_n Z_{n+1})^2 + (Y_{n+1} Z_n)^2) - 2YY_n Z_n Y_{n+1} Z_{n+1}, \end{aligned}$$

which increases the cost to $6\mathbf{M} + 3\mathbf{S}$. However, we will show in Section 3.1.6 that this can be optimized to give the exact same formula as the Montgomery curve differential addition from Section 3.1.3. Therefore it is obvious that the restriction to points of odd order is not required.

Other twisted Edwards differential addition formulae were derived in [82]. They make use of w-coordinates, that are defined as $w(x, y) = d(xy)^2$ for affine points $P = (x, y)$. Therefore, this method is not available in the context of SIDH, and we don't give more details on this here.

As described in [17], the j-invariant of a twisted Edwards curve is given by

$$\frac{16(a^2 + 14ad + d^2)^3}{ad(a-d)^4},$$

which shows that in this case, we need both parameters a and d for the computation of the j-invariant.

3.1.5 Switching between Montgomery and twisted Edwards curves

If we want to combine some of the ideas for Montgomery and twisted Edwards curves from above, we need an efficient way to switch between these two models. We slightly change our notation here in the following way: For a Montgomery curve over a field $\mathbb{K}$ with $A \in K \setminus \{-2, 2\}$ and $B \in K \setminus \{0\}$, we write

$$E_{M,A,B} : Bv^2 = u^3 + Au^2 + u.$$

For a twisted Edwards curve with distinct $a, d \in K \setminus \{0\}$ and $d \neq 1$, we write

$$E_{E,a,d} : ax^2 + y^2 = 1 + dx^2 y^2.$$

Then it is shown in [17] that $E_{E,a,d}$ is birationally equivalent to $E_{M,A,B}$, where

$$A = \frac{2(a+d)}{a-d} \quad \text{and} \quad B = \frac{4}{a-d},$$

and a birational equivalence is given by the map

$$(x, y) \mapsto (u, v) = \left(\frac{1+y}{1-y}, \frac{1+y}{(1-y)x}\right)$$

and its inverse

$$(u, v) \mapsto (x, y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right).$$

Note that these maps are not defined everywhere. For a way to handle exceptional points, we refer to [17].

However, if we use projective coordinates, particularly the XZ-only Montgomery arithmetic and the YZ-only twisted Edwards arithmetic, switching between these models is very simple. As seen in [48], a Montgomery point $(X_M : Z_M)$ can be transformed to the corresponding Edwards YZ-coordinates $(Y_E : Z_E)$ by the map

$$(X_M : Z_M) \mapsto (Y_E : Z_E) = (X_M - Z_M : X_M + Z_M).$$

A twisted Edwards point $(Y_E : Z_E)$ can be transformed to the corresponding Montgomery XZ-coordinates $(X_M : Z_M)$ by the map

$$(Y_E : Z_E) \mapsto (X_M : Z_M) = (Y_E + Z_E : Z_E - Y_E).$$

Therefore, switching between these two models costs only two additions.

3.1.6 Elliptic-curve arithmetic in SIDH

There are different stages of SIDH, where elliptic curve arithmetic takes place. In [63], all the arithmetic is done in XZ-only Montgomery coordinates.

3.1.6.1 Stage 1

In the key generation, Alice and Bob compute their respective secret kernel generator $P_i + [m_i]Q_i$ for some initially chosen points P_i and Q_i, and a random integer m_i. Thus, the starting curve parameters and full coordinates of P_i and Q_i are known.

3.1.6.2 Stage 2

During the computation of the isogenies, Alice and Bob have to compute several doublings or triplings, respectively. Since we work on different curves after each computed isogeny, the curve parameters are not fixed here. Furthermore, the points R_i that have to be doubled or tripled, are given in Montgomery XZ-coordinates.

3.1.6.3 Stage 3

In the computation of the shared secret, Alice and Bob again have to compute $\hat{P}_i + [m_i]\hat{Q}_i$ for their secret integer m_i and some received points $\hat{P}_i$ and $\hat{Q}_i$ from the public key. Here, only the normalized curve parameter A and the normalized X-coordinates of $\hat{P}_i$, $\hat{Q}_i$, and $\hat{P}_i - \hat{Q}_i$ are known.

3.1.7 Twisted Edwards curve arithmetic in SIDH

In this section, we analyze, how twisted Edwards curve arithmetic can be used in the different stages of SIDH.

Stage 1

The situation of Alice and Bob is equal here, except for the different initial points provided and the probably different random numbers. Therefore, we don't have to distinguish between the two parties here. In the implementation from [63], we start with the Montgomery curve $y^2 = x^3 + x$ and the computation of $[m]Q$ uses a Montgomery ladder. Per bit of the integer m, one doubling and one addition are performed in Montgomery XZ-only arithmetic. Since the computation entirely takes place in the basefield $\mathbb{F}_p$ and because of the choice of the curve parameters, the cost of this is $5\mathbf{M} + 4\mathbf{S}$ per bit of m.

As seen above, and mentioned in [48], we can replace each doubling of a point Q by a doubling in Edwards coordinates:

1. Compute the corresponding Edwards point Q_E,
2. Compute $[2]Q_E$ by a twisted Edwards doubling,
3. Switch back to Montgomery coordinates to obtain $[2]Q$.

However, in the context of SIDH, this can be optimized as follows: Since the Montgomery parameters of the starting curve are $A = 0$ and $B = 1$, the corresponding Edwards parameters are $a = 2$ and $d = -2$. Therefore, all multiplications by curve parameters can be replaced by additions. Plugging the parameters into the doubling formulae, we obtain

$$\begin{aligned} Y_2 &= 2Y^4 + 4Y^2Z^2 - 2Z^4, \\ Z_2 &= -2Y^4 + 4Y^2Z^2 + 2Z^4. \end{aligned}$$

Instead of computing the Edwards doubling and transforming back to Montgomery coordinates afterwards by computing $Y_2 + Z_2$ and $Z_2 - Y_2$, we can combine these steps, since

$$\begin{aligned} Y_2 + Z_2 &= 2Y^2Z^2, \\ Z_2 - Y_2 &= Z^4 - Y^4, \end{aligned}$$

so we do not have to compute Y_2 and Z_2 explicitly. Therefore, we save a few additions. Furthermore, we get the transformation to Edwards coordinates for free, since in each ladder step, a Montgomery differential addition is performed, during which the required values occur. In total, the Edwards doubling costs $5\mathbf{S}$ here, and the combined Montgomery differential addition and Edwards doubling, and hence one step in the ladder, costs $3\mathbf{M}+7\mathbf{S}$.

A different approach to compute a multiple $[m]Q$ of a point by Edwards arithmetic is given in the context of the elliptic-curve method for factorization in [18], where multiples of points are computed in full coordinates. The fastest way described there is the combination of the doubling in projective coordinates and the addition in extended coordinates that are stated above. Bernstein and Lange use 'signed sliding fractional window' addition-subtraction chains, which are defined in [22], that define the sequence of doublings and additions for a fast computation of $[m]Q$. Using such a chain, only one doubling and ε additions are required per bit of m, where ε converges to 0 for increasing bitlength of m. For Alice, the random integer m has a maximal bitlength of 372. We see from [18], that a bitlength in this magnitude requires approximately 0.99 doublings and 0.19 additions per bit. Thus, calculating with a bitlength of 372, we end up with roughly $1740\mathbf{M}+1473\mathbf{S}$ in total for the computation of $[m]Q$, ignoring further computations for switching between the coordinate models. In comparison, the ladder from [63] needs $5\mathbf{M}+4\mathbf{S}$ per bit of m, and therefore a total of $1860\mathbf{M}+1488\mathbf{S}$. However, since m is randomly chosen each time, we always have to compute a fast chain, and in addition, more storage is required for the chain and the saved intermediate values.

Stage 2

In this stage of the algorithm, the situation of Alice and Bob is slightly different, since they have to compute doublings or triplings, respectively. We first focus on Alice's computations, where doublings are needed. The first thing to note here is that in every step, we work on a new curve with different parameters, and therefore a multiplication by a curve coefficient costs as much as a general field multiplication $\mathbf{M}$, since we cannot expect the parameters to stay small. For a YZ- only Edwards doubling we thus need $4\mathbf{M} + 5\mathbf{S}$. A doubling in full projective coordinates would be slightly cheaper, using $4\mathbf{M}+4\mathbf{S}$, but the X-coordinates are not known here. Although it is clear, that compared to the Montgomery arithmetic from [63], the computation is more expensive this way, we show how it can be done.

For using these formulae, we first have to recover the corresponding Edwards parameters, in the following denoted by a_E and d_E. Since in the implementation of [63], the curve parameters are in projective form $(A : C)$, where the usual Montgomery parameter $a = A/C$, and b resp. B is discarded completely, it is not possible to use the formulae above directly to recover the Edwards parameters. However, we can rewrite them as

$$\frac{A}{C} = \frac{2(a_E + d_E)}{a_E - d_E} \quad \text{and} \quad \frac{B}{C} = \frac{4}{a_E - d_E}.$$

We can then fix $B = 1$ and thus obtain

$$a_E = A + 2C \quad \text{and} \quad d_E = A - 2C.$$

Therefore, all the doublings in this section of the algorithm can be done using Edwards coordinates in the following way:

1. Compute the corresponding Edwards point (cost: 2 additions)
2. Recover the Edwards parameters a_E and d_E (cost: 3 additions)
3. Compute all the required doublings (cost: $4\mathbf{M} + 5\mathbf{S}$ each)
4. Transform the resulting point back into Montgomery coordinates (cost: 2 additions)

On Bob's side, we have to compute point triplings. There are various ways to use Edwards curve arithmetic from section 4. We can simply perform an Edwards doubling followed by an Edwards differential addition to triple a point. Without further optimization, this takes $10\mathbf{M} + 8\mathbf{S}$. Now assume that we want to switch to Montgomery coordinates after the computation of the tripling. Given $P = (Y : Z)$ and $[2]P = (Y_2 : Z_2)$ in Edwards coordinates, we can compute $[3]P = (Y_3 : Z_3)$ by the formulae from section 4:

$$\begin{aligned} Y_3 &= -Y((YZ_2)^2 + (Y_2Z)^2) + 2YZ^2Y_2Z_2, \\ Z_3 &= Z((YZ_2)^2 + (Y_2Z)^2) - 2Y^2ZY_2Z_2. \end{aligned}$$

For the change to Montgomery coordinates, we have to compute

$$\begin{aligned} Y_3 + Z_3 &= (Z - Y)((YZ_2)^2 + (Y_2Z)^2) + 2(Z - Y)(YZY_2Z_2) \\ &= (Z - Y)(YZ_2 + Y_2Z)^2 \end{aligned}$$

for the Montgomery X-coordinate, and

$$\begin{aligned} Z_3 - Y_3 &= (Z + Y)((YZ_2)^2 + (Y_2Z)^2) - 2(Z + Y)(YZY_2Z_2) \\ &= (Z + Y)(YZ_2 - Y_2Z)^2 \end{aligned}$$

for the Montgomery Z-coordinate. This looks very similar to the Montgomery differential addition formulae. If we now interchange the Edwards coordinates with their corresponding Montgomery coordinates, we end up with the exact same formulae from Section 3.1.3. Therefore the twisted

Edwards differential addition from [134] is basically a less efficient representation of the Montgomery formulae, and thus we can restrict to the use of the Montgomery formulae here.

We can now combine the Edwards doubling with the Montgomery differential addition, leading to a cost of $8\mathbf{M} + 7\mathbf{S}$ per tripling. This is again more expensive than the tripling from [63], which costs $8\mathbf{M} + 4\mathbf{S}$. In [62] and [83], the cost was further reduced to $7\mathbf{M} + 5\mathbf{S}$.

Stage 3

Similar to stage 1, we want to compute $P + [m]Q$ here. However the circumstances are different here. We are given the public keys, namely the normalized X-coordinates of P, Q, and $P - Q$, and the normalized Montgomery curve parameter A, which is calculated from these values. In [63], the three-point-ladder from [109] is used, which computes one differential addition and one combined doubling and differential addition per step.
For the deployment of Edwards curves, we can again replace the doubling of the combined step by an Edwards doubling. However, in contrast to stage 1, we cannot expect the coefficients $a_E = A + 2$ and $d_E = A - 2$ to be small, so we have to count them as full multiplications here as well. This leads to an extra cost of $4\mathbf{M}$ compared to the computation in stage 1, and thus a cost of $7\mathbf{M} + 7\mathbf{S}$ per combined doubling and addition. A complete ladder step, which includes one more differential addition, costs $10\mathbf{M}+9\mathbf{S}$ in this case. In comparison, a ladder step in [63] costs only $9\mathbf{M}+6\mathbf{S}$.

As in stage 1, another option would be to compute $P + [m]Q$ in full Edwards projective and extended coordinates. However, we would need to recover the Y-coordinates of P and Q for that.

3.1.8 Implementation results

We implemented SIDH with an optimal strategy based on [63] in Python 2.7. This was used as a reference for performance comparisons to the described Edwards arithmetic. The Python scripts can be found on `https://github.com/sopmacF/hybrid-SIDH`. In the Edwards script, we implemented the Stages 1 and 2 from above, namely in stage 1 a basefield ladder, that uses a combination of an Edwards doubling and a Montgomery differential addition, and in stage 2 Edwards doublings and triplings as described. In total, the computational effort increased by roughly 10% with all these changes, including an adjusted optimal strategy.

We further implemented our changes in the C implementation of Costello, Longa, and Naehrig [63], which is available at [65]. In Table 3.1, we give

	CLN [63]	This work
Doublings	7,305	10,073
Triplings	14,503	17,640

Table 3.1: Performance comparison of doublings and triplings. All timings are given in clock cycles and were measured on a 2.5GHz Intel Core i7-6500 Skylake processor running Ubuntu 16.04 LTS.

Protocol Phase	CLN [63]	This work
Alice's Key Generation	30.2	33.7
Bob's Key Generation	34.1	36.8
Alice's Shared Secret	28.8	32.2
Bob's Shared Secret	32.8	35.5

Table 3.2: Performance comparison of the SIDH protocol phases. The running times are given in 10^6 clock cycles. The setup is the same as in Table 1.

a comparison of our Edwards-based doubling and tripling formulae from section 7.2 and the ones from [65].
Table 3.2 gives a comparison of the total cost of the key exchange phases for the respectively used formulae. Alice and Bob both use the basefield ladder from section 7.1 in our implementation, which is more efficient than the one from [63], whenever the cost of $\mathbf{S}$ is less than $\frac{2}{3}\mathbf{M}$.
As expected, our Edwards version is roughly 10% slower, like in our Python implementation. Our edited version of the implementation of [63], which also contains adjusted optimal strategies, is available at `https://github.com/sopmacF/hybrid-SIDH`.

3.1.9 Conclusion and future work

As a result we can suppose, that a hybrid SIDH scheme, which uses Edwards arithmetic whenever possible, does not yield an immediate speedup for SIDH. However, since the computations are almost as efficient as in the state-of-the-art implementation [63], it is still possible, that a full Edwards version of SIDH with efficient Edwards isogeny formulae can improve the performance of SIDH. In this case, if YZ-only arithmetic is used, it may be even advantageous to switch to Montgomery curves in some cases, e.g., to speed up doublings. However, we leave this question open for further investigation. In [10], the authors presented an implementation of SIDH for complete Edwards curves providing security benefits against side-channel attacks.

4 Constant-time Implementation

4.1 On Lions and Elligators: An efficient constant-time implementation of CSIDH

This chapter is for all practical purposes identical to the paper *On Lions and Elligators: An efficient constant-time implementation of CSIDH* [136] authored jointly with Michael Meyer and Steffen Reith, which was published at PQCrypto 2019.

4.1.1 Introduction

Isogeny-based cryptography is the most juvenile family of the current proposals for post-quantum cryptography. The first cryptosystem based on the hardness of finding an explicit isogeny between two given isogenous elliptic curves over a finite field was proposed in 1997 by Couveignes [66], eventually independently rediscovered by Rostovtsev and Stolbunov [164] in 2004, and therefore typically called CRS. Childs, Jao, and Soukharev [57] showed in 2010 that CRS can be broken using a subexponential quantum algorithm by solving an abelian hidden shift problem. To avoid this attack, Jao and De Feo [109] invented the new isogeny-based scheme SIDH (supersingular isogeny Diffie–Hellman) that works with supersingular curves over $\mathbb{F}_{p^2}$. SIKE [108] is a post-quantum key encapsulation mechanisms based on SIDH, which was submitted to the NIST post-quantum cryptography competition [178].

De Feo, Kieffer and Smith optimized CRS in 2018 [72]. Their ideas led to the development of CSIDH by Castryck, Lange, Martindale, Panny, and Renes [49], who adapted the CRS scheme to supersingular curves and isogenies defined over a prime field $\mathbb{F}_p$. They implemented the key exchange as a proof-of-concept, which is efficient, but does not run in constant time, and can therefore leak information about private keys. We note that building an efficient constant-time implementation of CSIDH is not as straightforward as in SIDH, where, speaking of running times, only one Montgomery ladder computation depends on the private key (see [63]).

In this chapter we present a constant-time implementation of CSIDH with many practical optimizations, requiring only a small overhead of factor 3.03 compared to the fastest variable-time implementation from [137].

4.1.2 CSIDH

We only cover the algorithmic aspects of CSIDH here, and refer to [49] for the mathematical background and a more detailed description.

Algorithm 2: Evaluating the class group action.

```
Input : $a \in \mathbb{F}_p$ such that $E_a : y^2 = x^3 + ax^2 + x$ is supersingular, and
        a list of integers $(e_1, ..., e_n)$ with $e_i \in \{-B, ..., B\}$ for all
        $i \leq n$.
Output: $a' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{a'}$.
1  while some $e_i \neq 0$ do
2      Sample a random $x \in \mathbb{F}_p$.
3      Set $s \leftarrow +1$ if $x^3 + ax^2 + x$ is a square in $\mathbb{F}_p$, else $s \leftarrow -1$.
4      Let $S = \{i \mid sign(e_i) = s\}$.
5      if $S = \varnothing$ then
6          Go to line 2.
7      $P = (x : 1)$, $k \leftarrow \prod_{i \in S} \ell_i$, $P \leftarrow [(p+1)/k]P$.   // $p = 4 \cdot \ell_1 \cdot ... \cdot \ell_n - 1$
8      foreach $i \in S$ do
9          $K \leftarrow [k/\ell_i]P$.
10         if $K \neq \infty$ then
11             Compute a degree-$\ell_i$ isogeny $\varphi : E_a \to E_{a'}$ with
                $ker(\varphi) = \langle K \rangle$.
12             $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $k \leftarrow k/\ell_i$, $e_i \leftarrow e_i - s$.
```

We first choose a prime of the form $p = 4 \cdot \ell_1 \cdot ... \cdot \ell_n - 1$, where the ℓ_i are small distinct odd primes. We work with supersingular curves over $\mathbb{F}_p$, which guarantees the existence of points of the orders ℓ_i, that enable us to compute ℓ_i-isogenies from kernel generator points by Vélu-type formulas [184].

A private key consists of a tuple $(e_1, ..., e_n)$, where the e_i are integers sampled from an interval $[-B, B]$. The absolute value $|e_i|$ specifies how many ℓ_i-isogenies have to be computed, and the sign of e_i determines, whether points on the current curve or on its twist have to be used as kernel generators. One can represent this graphically: Over $\mathbb{F}_p$, the supersingular ℓ_i-isogeny graph consists of distinct cycles. Therefore, we have to walk $|e_i|$ steps through the cycle for ℓ_i, and the sign of e_i tells us the direction.

Since this class group action is commutative, it allows a basic Diffie–Hellman-type key exchange: Starting from a supersingular curve E_0, Alice and Bob choose a private key as described above, and compute their public key curves E_A resp. E_B via isogenies, as described in Algorithm 2. Then Alice repeats her computations, this time starting at the curve E_B, and vice versa. Both parties then arrive at the same curve E_{AB}, which represents their shared secret. Furthermore, public keys can be verified efficiently in CSIDH (see [49]). Therefore, a static-static key-exchange is possible.

However, the quantum security is still an open problem. For our implementation we use CSIDH-512, the parameter set from [49], that is conjectured to satisfy NIST security level 1. In the light of the subexponential quantum attack on CRS and CSIDH [57], more analysis on CSIDH has been done in [30, 36, 23].

4.1.3 Leakage scenarios

It is clear and already mentioned in [49] that the proof-of-concept implementation of CSIDH is not side-channel resistant. In this chapter we focus on three scenarios that can leak information on the private key. Note that the second scenario features a stronger attacker. Further, there will of course be many more scenarios for side-channel attacks.

4.1.3.1 Timing leakage.

As the private key in CSIDH specifies how many isogenies of each degree have to be computed, it is obvious that this (up to additional effort for point multiplications due to the random choice of points) determines the running time of the algorithm. As stated in [137], the worst-case running time occurs for the private key $(5, 5, ..., 5)$, and takes more than 3 times as much as in the average case. The other extreme is the private key $(0, 0, ..., 0)$, which would require no computations at all. However, in a timing-attack-protected implementation, the running time should be independent from the private key.

4.1.3.2 Power analysis.

Instead of focusing on the running time, we now assume that an attacker can measure the power consumption of the algorithm. We further assume that from the measurement, the attacker can determine blocks which represent the two main primitives in CSIDH, namely point multiplication and isogeny computation, and can separate these from each other. Now assume that the attacker can separate the loop iterations from each other. Then the attacker can determine which private-key elements share the same sign from the isogeny blocks that are performed in the same loop, since they have variable running time based on the isogeny degree. This significantly reduces the possible key space and therefore also the complexity of finding the correct key.

4.1.3.3 Cache timing attacks.

In general, data flow from the secret key to branch conditions and array indices must be avoided in order to achieve protection against cache timing attacks [20]. Our implementation follows these guidelines to avoid vulnerabilities against the respective possible attacks.

4.1.4 Mitigating Leakage

In this section we give some ideas on how to fix these possible leakages in an implementation of CSIDH. We outline the most important ideas here, and give details about how to implement them efficiently in CSIDH-512 in Section 4.1.5.

4.1.4.1 Dummy isogenies.

First, it seems obvious that one should compute a constant number of isogenies of each degree ℓ_i, and only use the results of those required by the private key, in order to obtain a constant running time. However, in this case additional multiplications are required, if normal isogenies and unused isogenies are computed in the same loop[12]. We adapt the idea of using dummy isogenies from [137] for that cause. There it is proposed to design dummy isogenies, which instead of updating the curve parameters and evaluating the curve point P, compute $[\ell_i]P$ in the degree-ℓ_i dummy isogeny. Since the isogeny algorithm computes $[\frac{\ell_i-1}{2}]K$ for the kernel generator K, one can replace K by P there, and perform two more differential additions to compute $[\ell_i]P$. The curve parameters remain unchanged.

In consequence, a dummy isogeny simply performs a scalar multiplication. Therefore, the output point $[\ell_i]P$ then has order not divisible by ℓ_i, which is important for using this point to compute correct kernel generators in following iterations. Further, one can design the isogeny and dummy isogeny algorithms for a given degree ℓ_i such that they perform the same number and sequence of operations with only minor computational overhead compared to the isogenies from [137]. This is important to make it hard for side-channel attackers to distinguish between those two cases, since conditional branching can be avoided with rather small overhead.

[12]This is required, since otherwise, an attacker in the second leakage scenario can determine the private key easily.

4.1.4.2 Balanced vs. unbalanced private keys.

Using dummy isogenies to spend a fixed time on isogeny computations is not enough for a constant-time implementation, however. Another problem lies in the scalar multiplications in line 7 and line 9 of Algorithm 2. We use an observation from [137] to illustrate this. They consider the private keys $(5, 5, 5, ...)$ and $(5, -5, 5, -5, ...)$ and observe that for the first key, the running time is 50% higher than for the second key. The reason for this is that in the first case in order to compute one isogeny of each degree, the multiplication in line 7 is only a multiplication by 4, and the multiplication in line 9 has a factor of bitlength 509 in the first iteration, 500 in the second iteration, and so on.

For the second key, we have to perform one loop through the odd i and one through the even i in order two compute one isogeny of each degree ℓ_i. Therefore, the multiplications in line 7 are by 254 resp. 259 bit factors, while the bitlengths of the factors in the multiplications in line 9 are 252, 244,..., resp. 257, 248, and so on (see Figure 1). In total, adding up the bitlengths of all factors, we can measure the cost of all point multiplications for the computation of one isogeny per degree, where we assume that the condition in line 10 of Algorithm 1 never fails, since one Montgomery ladder step is performed per bit. For the first key, we end up with 16813 bits, while for the second key we only have 9066 bits.

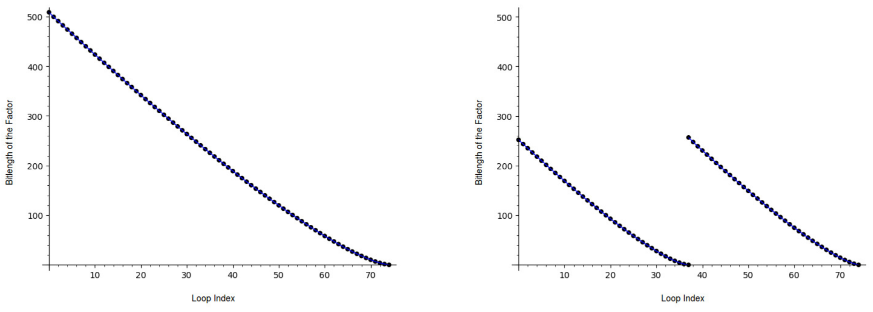


Figure 4.1: Bitlengths of factors for computing one isogeny per degree for the keys $(5, 5, ..., 5)$ (left) and $(5, -5, 5, -5, ...)$ (right).

This can be generalized to any private key: The more the key elements (or the products of the respective ℓ_i) are unbalanced, i.e., many of them share the same sign, the more the computational effort grows, compared to the perfectly balanced case from above. This behavior depends on the private key and can therefore leak information. Hence, it is clear that we have to prevent this in order to achieve a constant-time implementation.

One way to achieve this is to use constant-time Montgomery ladders that always run to the maximum bitlength, no matter how large the respective factor is. However, this would lead to a massive increase in running time. Another possibility for handling this is to only choose key elements of a fixed sign. Then we have to adjust the interval from which we sample the integer key elements, e.g. from $[-5, 5]$ to $[0, 10]$ in CSIDH-512. This however doubles the computational effort for isogenies (combined normal and dummy isogenies). We will return to this idea later.

4.1.4.3 Determining the sign distribution.

In our second leakage scenario, an attacker might determine the sign distribution of the key elements by identifying blocks of isogeny resp. dummy isogeny computations. One way of mitigating this attack would be to let each degree-ℓ_i isogeny run as long as a ℓ_{max}-isogeny, where ℓ_{max} is the largest ℓ_i. As used in [23], this is possible because of the Matryoshka-doll structure of the isogeny algorithms. This would allow an attacker in the second leakage scenario to only determine the number of positive resp. negative elements, but not their distribution, at the cost of a large increase of computational effort. We can also again restrict to the case that we only choose nonnegative (resp. only nonpositive) key elements. Then there is no risk of leaking information about the sign distribution of the elements, since in this setting the attacker knows this beforehand, at the cost of twice as many isogeny computations.

4.1.4.4 Limitation to nonnegative key elements.

Since this choice eliminates both of the aforementioned possible leakages, we use the mentioned different interval to sample private key elements from. In CSIDH-512, this means using the interval $[0, 10]$ instead of $[-5, 5]$. One might ask if this affects the security properties of CSIDH. As before, there are 11^{74} different tuples to choose from in CSIDH-512. In [49], the authors argue that there are multiple vectors $(e_1, e_2, ..., e_n)$, which represent the same ideal class, meaning that the respective keys are equivalent. However, they assume by heuristic arguments that the number of short representations per ideal class is small, i.e. the 11^{74} different keys $(e_1, e_2, ..., e_n)$, where all e_i are sampled from the interval $[-5, 5]$, represent not much less than 11^{74} ideal classes. If we now have two equivalent keys $e \neq f$ sampled from $[-5, 5]$, then we have a collision for our shifted interval as well, since shifting all elements of e and f by $+5$ results in equivalent keys $e' \neq f'$ with elements in $[0, 10]$, and vice versa. Therefore, our shifted version is equivalent to CSIDH-512 as defined in [49][13].

[13] One could also think of using the starting curve E', which is the result of applying the key $(5, 5, ..., 5)$ to the curve E_0. Then for a class group action evaluation using key elements from $[-5, 5]$ and the starting curve E' is equivalent to using key elements from $[0, 10]$ and the starting curve E_0.

In the following sections we focus on optimized implementations, using the mentioned countermeasures against attacks, i.e., sampling key elements from the interval $[0, 10]$ and using dummy isogenies.

4.1.5 Efficient Implementation

4.1.5.1 Straightforward Implementation

First, we describe the straightforward implementation of the evaluation of the class group action in CSIDH-512 with the choices from above, before applying various optimizations. We briefly go through the implementation aspects of the main primitives, i.e., scalar multiplications, isogenies and dummy isogenies, and explain why this algorithm runs in constant time, and does not leak information about the private key.

Algorithm 3: Constant-time evaluation of the class group action in CSIDH-512.

```
Input  : $a \in \mathbb{F}_p$ such that $E_a : y^2 = x^3 + ax^2 + x$ is supersingular, and
         a list of integers $(e_1, ..., e_n)$ with $e_i \in \{0, 1, .., 10\}$ for all
         $i \leq n$.
Output: $a' \in \mathbb{F}_p$, the curve parameter of the resulting curve $E_{a'}$.
1  Initialize $k = 4$, $e = (e_1, ..., e_n)$ and $f = (f_1, ..., f_n)$, where $f_i = 10 - e_i$.
2  while some $e_i \neq 0$ or $f_i \neq 0$ do
3      Sample random values $x \in \mathbb{F}_p$ until we have some $x$ where
        $x^3 + ax^2 + x$ is a square in $\mathbb{F}_p$.
4      Set $P = (x : 1)$, $P \leftarrow [k]P$, $S = \{i \mid e_i \neq 0$ or $f_i \neq 0\}$.
5      foreach $i \in S$ do
6          Let $m = \prod_{j \in S, j > i} \ell_j$.
7          Set $K \leftarrow [m]P$.
8          if $K \neq \infty$ then
9              if $e_i \neq 0$ then
10                 Compute a degree-$\ell_i$ isogeny $\varphi : E_a \to E_{a'}$ with
                    $ker(\varphi) = \langle K \rangle$.
11                 $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_i \leftarrow e_i - 1$.
12             else
13                 Compute a degree-$\ell_i$ dummy isogeny:
14                 $a \leftarrow a$, $P \leftarrow [\ell_i]P$, $f_i \leftarrow f_i - 1$.
15             if $e_i = 0$ and $f_i = 0$ then
16                 Set $k \leftarrow k \cdot \ell_i$.
```

4.1.5.2 Parameters.

As described in [49], we have a prime number $p = 4 \cdot \ell_1 \cdot \ell_2 \cdot ... \cdot \ell_n - 1$, where the ℓ_i are small distinct odd primes. We further assume that we have $\ell_1 > \ell_2 > ... > \ell_n$. In CSIDH-512 we have $n = 74$, and we sample the elements of private keys $(e_1, e_2, ..., e_n)$ from $[0, 10]$.

4.1.5.3 Handling the private key.

Similar to the original implementation of [49], we copy the elements of the private key into an array $e = (e_1, e_2, ..., e_n)$, where e_i determines how many isogenies of degree ℓ_i we have to compute. Furthermore, we set up another array $f = (10-e_1, 10-e_2, ..., 10-e_n)$, to determine how many dummy isogenies of each degree we have to compute. As we go through the algorithm, we compute all the required isogenies and dummy isogenies, reducing e_i resp. f_i by 1 after each degree-ℓ_i isogeny resp. dummy isogeny. We therefore end up with a total of 10 isogeny computations (counting isogenies and dummy isogenies) for each ℓ_i.

4.1.5.4 Sampling random points.

In Algorithm 3 line 3, we have to find curve points on the current curve that are defined over $\mathbb{F}_p$ instead of $\mathbb{F}_{p^2} \backslash \mathbb{F}_p$. As in [49] this can be done by sampling a random $x \in \mathbb{F}_p$, and computing y^2 by the curve equation $y^2 = x^3 + ax^2 + x$. We then check if y is defined over $\mathbb{F}_p$ by a Legendre symbol computation, i.e. by checking if $(y^2)^{(p-1)/2} \equiv 1 \pmod{p}$. If this is not the case, we simply repeat this procedure until we find a suitable point. Note that we require the curve parameter a to be in affine form. Since a will typically be in projective form after isogeny computations, we therefore have to compute the affine parameter each time before sampling a new point.

4.1.5.5 Elliptic curve scalar multiplications.

Since we work with Montgomery curves, using only projective XZ-coordinates, and projective curve parameters $a = A/C$, we can use the standard Montgomery ladder as introduced in [140], adapted to projective curve parameters as in [63]. This means that per bit of the factor, one combined doubling and differential addition is performed.

4.1.5.6 Isogenies.

For the computation of isogenies, we use the formulas presented in [137]. They combine the Montgomery isogeny formulas by Costello and Hisil [62], and Renes [161] with the twisted Edwards formulas by Moody and Shumow [141], in order to obtain an efficient algorithm for the isogeny computations in CSIDH. For a ℓ_i-isogeny, this requires a point K of order ℓ_i as kernel generator, and the projective parameters A and C of the current curve. It outputs the image curve parameters A' and C', and the evaluation of the point P. As mentioned before, the algorithm computes all multiples of the point K up to the factor $\frac{\ell_i-1}{2}$. See, e.g., [23] for more details.

4.1.5.7 Dummy isogenies.

As described before, we want the degree-ℓ_i dummy isogenies to output the scalar multiple $[\ell_i]P$ instead of an isogeny evaluation of P. Therefore, we interchange the points K and P in the original isogeny algorithm, such that it computes $[\frac{\ell_i-1}{2}]P$. We then perform two more differential additions, i.e. compute $[\frac{\ell_i+1}{2}]P$ from $[\frac{\ell_i-1}{2}]P$, P, and $[\frac{\ell_i-3}{2}]P$, and compute $[\ell_i]P$ from $[\frac{\ell_i+1}{2}]P$, $[\frac{\ell_i-1}{2}]P$, and P.

As mentioned before, we want isogenies and dummy isogenies of degree ℓ_i to share the same code in order to avoid conditional branching. Hence, the two extra differential additions are also performed in the isogeny algorithm, without using the results. In our implementation, a conditional point swapping based on a bitmask ensures that the correct input point is chosen. This avoids conditional branching that depends on the private key in line 9 of Algorithm 3 (and lines 11 and line 27 of Algorithm 6).

If one is concerned that a side-channel attacker can detect that the curve parameters A and C are not changed for some time (meaning that a series of dummy isogenies is performed), one could further re-randomize the projective representation of the curve parameter A/C by multiplying A and C by the same random number[14] $1 < \alpha < p$.

4.1.5.8 Running time

We now explain why this algorithm runs in constant time. As already explained, we perform 10 isogeny computations (counting isogenies and dummy isogenies) for each degree ℓ_i. Furthermore, isogenies and dummy isogenies have the same running time, since they share the same code, and conditionally branching is avoided. Therefore the total computational effort for isogenies is constant, independent from the respective private key. We also set the same condition (line 8 of Algorithm 3) for the kernel generator for the computation of a dummy isogeny, in order not to leak information.

Sampling random points and finding a suitable one doesn't run in constant time in Algorithm 3. However, the running time only depends on randomly chosen values, and does not leak any information on the private key.

Now for simplicity assume that we always find a point of full order, i.e. a point that can be used to compute one isogeny of each degree ℓ_i. Then it

[14]One could actually use an intermediate value $\alpha \in \mathbb{F}_p \backslash \{0, 1\}$ of the isogeny computation, since the factor is not required to be truly random.

is easy to see that the total computational effort for scalar multiplications in Algorithm 2 is constant, independent from the respective private key. If we now allow random points, we will typically not satisfy the condition in line 8 of Algorithm 2 for all i. Therefore, additional computations (sampling random points, and scalar multiplications) are required. However, this does not leak information about the private key, since this only depends on the random choice of curve points, but not on the private key.

Hence, we conclude that the implementation of Algorithm 3 as described here prevents the leakage scenarios considered in Section 4.1.3. It is however quite slow compared to the performance of variable-time CSIDH-512 in [137, 49]. In the following section, we focus on how to optimize and speed up the implementation.

4.1.5.9 Optimizations

4.1.5.10 Sampling points with Elligator.

In [23] Bernstein, Lange, Martindale, and Panny pointed out that Elligator [21], specifically the Elligator 2 map, can be used in CSIDH to be able to choose points over the required field of definition. Since we only need points defined over $\mathbb{F}_p$, this is especially advantageous in our situation. For $a \neq 0$ the Elligator 2 map works as follows (see [23]):

- Sample a random $u \in \{2, 3, ..., (p-1)/2\}$.
- Compute $v = a/(u^2 - 1)$.
- Compute e, the Legendre symbol of $v^3 + av^2 + v$.
- If $e = 1$, output v. Otherwise, output $-v - a$.

Therefore, for all $a \neq 0$, we can replace the search for a suitable point in line 3 of Algorithm 3, at the cost of an extra inversion. However, as explained in [23], one can precompute $1/(u^2 - 1)$ for some values of u, e.g. for $u \in \{2, 3, 4, ...\}$. Then the cost is essentially the same as for the random choice of points, but we always find a suitable point this way, compared to the probability of 1/2 when sampling random points. This could, however, potentially lead to the case that we cannot finish the computation: Consider that we only have one isogeny of degree ℓ_i left to compute, but for all of the precomputed values of u, the order of the corresponding point is not divided by ℓ_i. Then we would have to go back to a random choice of points to finish the computation. However, our experiments suggest that it is enough to have 10 precomputed values. Note that the probability for actually finding points of suitable order appears to be almost unchanged when using Elligator instead of random points, as discussed in [23].

For $a = 0$, [23] also show how to adapt the Elligator 2 map to this case, but also argue that one could precompute a point of full order (or almost full order, i.e. divided by all ℓ_i) and simply use this point whenever $a = 0$. We follow their latter approach.

4.1.5.11 SIMBA (Splitting isogeny computations into multiple batches).

In Section 4.1.4, we analyzed the running time of variable-time CSIDH-512 for the keys $e_1 = (5, 5, ..., 5)$ and $e_2 = (5, -5, 5, -5, ...)$. For the latter, the algorithm is significantly faster, because of the smaller multiplications during the loop (line 9 of Algorithm 1), see Figure 1. We adapt and generalize this observation here, in order to speed up our constant-time implementation.

Consider for our setting the key $(10, 10, ..., 10)$ and that we can again always choose points of full order. To split the indices into two sets (exactly as Algorithm 1 does for the key e_2), we define the sets $S_1 = \{1, 3, 5, ..., 73\}$ and $S_2 = \{2, 4, 6, ..., 74\}$. Then the loops through the ℓ_i for $i \in S_1$ resp. $i \in S_2$ require significantly smaller multiplications, while only requiring to compute $[4k]P$ with $k = \prod_{i \in S_2} \ell_i$ resp. $k = \prod_{i \in S_1} \ell_i$ beforehand. We now simply perform 10 loops for each set, and hence this gives exactly the same speedup over Algorithm 2, as Algorithm 1 gives for the key e_2 compared to e_1, by using two batches of indices instead of only one.

One might ask if splitting the indices in two sets already gives the best speedup. We generalize the observation from above, now splitting the indices into m batches, where $S_1 = \{1, m+1, 2m+1, ...\}$, $S_2 = \{2, m+2, 2m+2, ...\}$, and so on[15]. Before starting a loop through the indices $i \in S_j$ with $1 \leq j \leq m$, one now has to compute $[4k]P$ with $k = \prod_{h \notin S_j} \ell_h$. The number and size of these multiplications grows when m grows, so we can expect that the speedup turns into an increasing computational effort when m is too large.

To find the best choice for m, we computed the total number of Montgomery ladder steps during the computation of one isogeny of each degree in CSIDH-512 for different m, with the assumptions from above. We did not take into account here that when m grows, we will have to sample more points (which costs at least one Legendre symbol computation each), since this depends on the cost ratio between Montgomery ladder steps and Legendre symbol computations in the respective implementation. Table 1 shows that the optimal choice should be around $m = 5$.

If we now come back to the choice of points through Elligator, the assumption from above does not hold anymore, and with very high probability, we will need more than 10 loops per index set. Typically, soon after 10 loops through each batch the large degree isogenies will be finished, while there are some small-degree isogenies left to compute. In this case our optimization backfires, since in this construction, the indices of the missing ℓ_i will

[15]Note that in [23] a similar idea is described. However, in their algorithm only two isogeny degrees are covered in each iteration. Our construction makes use of the fact that we restrict to intervals of nonnegative numbers for sampling the private key elements.

Table 4.1: Number of Montgomery ladder steps for computing one isogeny of each degree in CSIDH-512 for different numbers of batches m.

m	1	2	3	4	5	6	7
Ladder steps	16813	9066	6821	5959	5640	5602	5721

be distributed among the m different batches. We therefore need large multiplications in order to only check a few small degrees per set. Hence it is beneficial to define a number $\mu \geq 10$, and merge the batches after μ steps, i.e., simply going back to Algorithm 2 for the computation of the remaining isogenies. We dub this construction SIMBA-m-μ.

4.1.5.12 Sampling private-key elements from different intervals.

Instead of sampling all private-key elements from the interval $[0, 10]$, and in total computing 10 isogenies of each degree, one could also consider to choose the key elements from different intervals for each isogeny degree, as done in [72]. For a private key $e = (e_1, e_2, ..., e_n)$, we can choose an interval $[0, B_i]$ for each e_i, in order to e.g. reduce the number of expensive large-degree isogenies at the cost of computing more small-degree isogenies. We require $\prod_i (B_i + 1) \approx 11^{74}$, in order to obtain the same security level as before. For the security implication of this choice, similar arguments as in Section 4.1.4 apply.
Trying to find the optimal parameters B_i leads to a large integer optimization problem, which is not likely to be solvable exactly. Therefore, we heuristically searched for parameters likely to improve the performance of CSIDH-512. We present them in Section 4.1.6 and Appendix.

Note that if we choose $B = (B_i, ..., B_n)$ differently from $B = (10, 10, ..., 10)$, the benefit of our optimizations above will change accordingly. Therefore, we changed the parameters m and μ in our implementation according to the respective B.

4.1.5.13 Skip point evaluations.

As described before, the isogeny algorithms compute the image curve parameters, and push a point P through the isogeny. However, in the last isogeny per loop, this is unnecessary, since we choose a new point after the isogeny computation anyway. Therefore, it saves some computational effort, if we skip the point-evaluation part in these cases.

4.1.5.14 Application to variable-time CSIDH.

Note that many of the optimizations from above are also applicable to variable-time CSIDH-512 implementations as in [137] or [49]. We could therefore also speed up the respective implementation results using the mentioned methods.

4.1.6 Implementation Results

We implemented our optimized constant-time algorithm in C, using the implementation accompanying [137], which is based on the implementation from the original CSIDH paper by Castryck, Lange, Martindale, Panny, and Renes [49]. For example the implementation of the field arithmetic in assembly is the one from [49]. Our final algorithm, containing all the optimizations from above, can be found in Appendix.

Since we described different optimizations that can influence one another, it is not straightforward to decide which parameters B, m, and μ to use. Therefore, we tested various choices and combinations of parameters B, m, and μ, assuming $\ell_1 > \ell_2 > ... > \ell_n$. The parameters and implementation results can be found in Appendix. The best parameters we found are given by

$$\begin{aligned}B = [&5,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,11,11,11,\\&11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,13,13,13,13,13,13,13\\&13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13]\end{aligned}$$

using SIMBA-5-11, where the key element e_i is chosen from $[0, B_i]$. We do not claim that these are the optimal parameters; there might be better choices that we did not consider in our experiments.

We further tried to rearrange the order of the primes ℓ_i in the different loops. As pointed out in [137], it is beneficial to go through the ℓ_i in descending order. However, if we suppress isogeny point evaluations in the last iteration per loop, this means that these savings refer to small ℓ_i, and therefore the impact of this is rather small. Hence, we put a few large primes at the end of the loops, therefore requiring more computational effort for point multiplications, which is however in some situations outweighed by the larger savings from not evaluating points.

In this way, the best combination we found for CSIDH-512 is $\ell_1 = 349$, $\ell_2 = 347$, $\ell_3 = 337$,..., $\ell_{69} = 3$, $\ell_{70} = 587$, $\ell_{71} = 373$, $\ell_{72} = 367$, $\ell_{73} = 359$, and $\ell_{74} = 353$, using SIMBA-5-11 and B from above, where the B_i are swapped accordingly to the ℓ_i.

In Table 2, we give the cycle count and running time for the implementation using the parameters from above. The code is freely available at `https://github.com/sopmacF/On-Lions-and-Elligators` and `https://doi.org/10.5281/zenodo.6900027`.

To give a comparison that mainly shows the impact of SIMBA and the different choice of B, we also ran the straightforward implementation according to Algorithm 2 with $B = [10, 10, ..., 10]$, also using Elligator. In this case, we measured 621.5 million clock cycles in the same setting as above.

Table 4.2: Performance of one class-group-action evaluation in CSIDH-512 with the mentioned parameters. All timings were measured on an Intel Core i7-6500 Skylake processor running Ubuntu 16.04 LTS, averaged over 1 000 runs.

Clock Cycles $\times 10^8$	wall clock time
3.145	121.3 ms

Compared to the performance of the variable-time implementation from [137], the results from Table 2 mean a slowdown of factor 3.03. However, as mentioned, also the variable-time implementation can benefit from the optimizations from this paper, so this comparison should not be taken too serious.

4.1.7 Conclusion

We present the first implementation of CSIDH that prevents certain side-channel attacks, such as timing leakages. However, there might be more leakage models, depending on how powerful the attacker is. There is also more work to be done on making this implementation as efficient as possible. It may, e.g., be possible to find a CSIDH-friendly prime p that allows for faster computations in $\mathbb{F}_p$.

Also the security features of CSIDH remain an open problem. More analysis on this is required, to show if the parameters are chosen correctly for the respective security levels.

We note that our results depend on the parameters from CSIDH-512. However, it is clear that the described optimizations can be adapted to other parameter sets and security levels as well.

4.1.8 Appendix

4.1.8.1 Implementation Results

We tested several parameters in a dynamical implementation, as explained in the chapter. The setting is the same as in Section 4.1.6. For the parameters $B^0, ..., B^4$ we chose

$$B^0 = [10, 10, 10, ..., 10],$$

$$\begin{aligned} B^1 = [&1,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,8,8,8,8,8,8,8,\\ &12,12,12,12,12,12,12,12,12,12,12,12,12,12,14,14,14,14,14,\\ &14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,\\ &14,14,14,14,14,14,14,14],\end{aligned}$$

	1st		2nd		3rd	
B^0	μ=10 m=5	338.1	μ=10 m=6	343.5	μ=11 m=5	343.7
B^1	μ=12 m=4	329.3	μ=14 m=4	330.6	μ=13 m=4	330.8
B^2	μ=11 m=5	326.5	μ=12 m=5	327.0	μ=11 m=4	327.6
B^3	μ=16 m=4	333.8	μ=17 m=4	337.6	μ=16 m=3	339.3
B^4	μ=20 m=3	397.5	μ=20 m=4	399.0	μ=21 m=3	399.5

$$\begin{aligned}
B^2 = [&5,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,\\
&11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,11,13,\\
&13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,\\
&13,13,13,13,13,13,13,13],
\end{aligned}$$

$$\begin{aligned}
B^3 = [&2,4,4,4,4,4,4,4,4,4,4,4,4,4,6,6,6,6,6,6,6,6,6,6,6,10,10,\\
&10,10,10,10,10,10,10,10,10,16,16,16,16,16,16,16,16,16,16,\\
&16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,\\
&16,16,16,16,16,16,16,16,16], \text{ and}
\end{aligned}$$

$$\begin{aligned}
B^4 = [&2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,12,20,20,\\
&20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,\\
&20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,\\
&20,20,20,20,20,20,20,20,20].
\end{aligned}$$

We measured many different combinations with different m and μ, running SIMBA-m-μ as described above, averaging the running time over 100 runs per parameter set, given in 10^6 clock cycles. For each B^i, we present the three best combinations we found.

For the best combinations mentioned above, we further tried to rearrange the order of the primes ℓ_i in the loops. As pointed out in [137], it is beneficial to go through the ℓ_i in descending order. However, if we suppress isogeny point evaluations in the last iteration per loop, this means that these savings refer to small ℓ_i, and therefore the impact of this is rather small. Hence, we put a few large primes at the end of the loops, therefore requiring more computational effort for point multiplications, which is however in some situations outweighed by the larger savings from not evaluating points.

In this way, the best combination we found for CSIDH-512 is $\ell_1 = 349$, $\ell_2 = 347$, $\ell_3 = 337$,..., $\ell_{69} = 3$, $\ell_{70} = 587$, $\ell_{71} = 373$, $\ell_{72} = 367$, $\ell_{73} = 359$, and

$\ell_{74} = 353$, using SIMBA-5-11 and B^2, where the B_i are swapped accordingly to the ℓ_i.

In the same setting as in Section 4.1.7, we measured 322.6 million clock cycles for this combination, which saves 3.9 million clock cycles compared to the results from above.

4.1.8.2 Algorithms

In this section we describe our constant-time algorithm, containing the optimizations from above. We split the application of SIMBA in two parts: SIMBA-I splits the isogeny computations in m batches, and SIMBA-II merges them after μ rounds. Note that in our implementation, it is actually not required to generate all the arrays from SIMBA-I.

Algorithm 5 shows the full class group action evaluation. Due to many loops and indices, it looks rather complicated. We recommend to additionally have a look at our implementation, provided in Section 4.1.6.

Algorithm 4: SIMBA-I.

```
Input  : $e = (e_1, ..., e_n)$, $B = (B_1, ..., B_n)$, $m$.
Output: $e^{i,iso} = (e_1^{i,iso}, ..., e_n^{i,iso})$, $e^{i,dum} = (e_1^{i,dum}, ..., e_n^{i,dum})$, $k_i$ for
        $i \in \{0, ..., m-1\}$.
1 Initialize $e^{i,iso} = e^{i,dum} = (0, 0, ..., 0)$ and $k_i = 4$ for $i \in \{0, ..., m-1\}$
2 foreach $i \in \{1, ..., 74\}$ do
3     $e_i^{i\%m,iso} \leftarrow e_i$.
4     $e_i^{i\%m,dum} \leftarrow B_i - e_i$.
5     foreach $j \in \{1, ..., m\}$ do
6         if $j \neq (i\%m)$ then
7             $k_i \leftarrow k_i \cdot \ell_i$.
```

Algorithm 5: SIMBA-II.

```
Input  : $e^{i,iso} = (e_1^{i,iso}, ..., e_n^{i,iso})$ and $e^{i,dum} = (e_1^{i,dum}, ..., e_n^{i,dum})$ for
         $i \in \{0, ..., m-1\}$, $m$.
Output: $e^{iso} = (e_1^{iso}, ..., e_n^{iso})$, $e^{dum} = (e_1^{dum}, ..., e_n^{dum})$, and $k$.
1 Initialize $e^{iso} = e^{dum} = (0, 0, ..., 0)$, and $k = 4$.
2 foreach $i \in \{1, ..., 74\}$ do
3     $e_i^{iso} \leftarrow e_i^{i\%m,iso}$.
4     $e_i^{dum} \leftarrow e_i^{i\%m,dum}$.
5     if $e_i^{iso} = 0$ and $e_i^{dum} = 0$ then
6         $k \leftarrow k \cdot \ell_i$.
```

Algorithm 6: Constant-time evaluation of the class group action in CSIDH-512.

```
Input  : $a \in \mathbb{F}_p$ , $e = (e_1, ..., e_n)$, $B$, $m$, $\mu$.
Output: $a'$ such that $[\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n}]E_a = E_{a'}$.
1  Run SIMBA-I($e$, $B$, $m$).
2  foreach $i \in \{1, ..., \mu\}$ do
3      foreach $j \in \{1, ..., m\}$ do
4          Run Elligator to find a point $P$, where $y_P \in \mathbb{F}_p$.
5          $P \leftarrow [k_j]P$.
6          $S = \{\iota \mid e_\iota^{m,iso} \neq 0$ or $e_\iota^{m,dum} \neq 0\}$.
7          foreach $\iota \in S$ do
8              $\alpha = \prod_{\kappa \in S, \kappa > \iota} \ell_\kappa$.
9              $K \leftarrow [\alpha]P$.
10             if $K \neq \infty$ then
11                 if $e_\iota^{j,iso} \neq 0$ then
12                     Compute a degree-$\ell_\iota$ isogeny $\varphi : E_a \to E_{a'}$ with
                        $ker(\varphi) = \langle K \rangle$.
13                     $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_\iota^{j,iso} \leftarrow e_\iota^{j,iso} - 1$.
14                 else
15                     Compute a degree-$\ell_\iota$ dummy isogeny:
16                     $a \leftarrow a$, $P \leftarrow [\ell_\iota]P$, $e_\iota^{j,dum} \leftarrow e_\iota^{j,dum} - 1$.
17                 if $e_\iota^{j,iso} = 0$ and $e_\iota^{j,dum} = 0$ then
18                     Set $k_j = k_j \cdot \ell_\iota$.
19 Run SIMBA-II($e^{i,iso}$ and $e^{i,dum}$ for $i \in \{0, ..., m-1\}$, $m$).
20 while some $e_i^{iso} \neq 0$ or $e_i^{dum} \neq 0$ do
21     Run Elligator to find a point $P$, where $y_P \in \mathbb{F}_p$.
22     Set $P = (x : 1)$, $P \leftarrow [k]P$, $S = \{i \mid e_i^{iso} \neq 0$ or $e_i^{dum} \neq 0\}$.
23     foreach $i \in S$ do
24         Let $m = \prod_{j \in S, j < i} \ell_i$.
25         Set $K \leftarrow [m]P$.
26         if $K \neq \infty$ then
27             if $e_i^{iso} \neq 0$ then
28                 Compute a degree-$\ell_i$ isogeny $\varphi : E_a \to E_{a'}$ with
                    $ker(\varphi) = \langle K \rangle$.
29                 $a \leftarrow a'$, $P \leftarrow \varphi(P)$, $e_i^{iso} \leftarrow e_i^{iso} - 1$.
30             else
31                 Compute a degree-$\ell_i$ dummy isogeny:
32                 $a \leftarrow a$, $P \leftarrow [\ell_i]P$, $e_i^{dum} \leftarrow e_i^{dum} - 1$.
33             if $e_i^{iso} = 0$ and $e_i^{dum} = 0$ then
34                 Set $k = k \cdot \ell_i$.
```

4.2 CTIDH: faster constant-time CSIDH

This chapter is for all practical purposes identical to the paper *CTIDH: faster constant-time CSIDH* [12] authored jointly with Gustavo Banegas, Daniel J. Bernstein, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková, which was published at CHES 2021.

4.2.1 Introduction

Isogeny-based cryptography, a relatively new area of post-quantum cryptography, has gained substantial attention in the past few years. Schemes like SIDH (Supersingular Isogeny Diffie–Hellman) [109] and CSIDH (Commutative Supersingular Isogeny Diffie–Hellman) [49] offer key-exchange protocols with the smallest key sizes among post-quantum systems. CSIDH is even a non-interactive key exchange, matching the data flow of traditional Diffie–Hellman key exchange, and it has received a considerable amount of attention related to constant-time algorithms [136, 154, 51, 104, 56, 3, 54].

Briefly, one can explain CSIDH as follows. Pick small odd primes $\ell_1 < \ell_2 < \cdots < \ell_n$ such that $p = 4 \cdot \prod_{i=1}^{n} \ell_i - 1$ is also prime. A public key is a supersingular elliptic curve $E_A/\mathbb{F}_p : y^2 = x^3 + Ax^2 + x$, specified by a single element $A \in \mathbb{F}_p$. Given this curve one can efficiently compute two curves ℓ_i-isogenous to E_A, denoted $\mathfrak{l}_i \star E_A$ and $\mathfrak{l}_i^{-1} \star E_A$, for any of the ℓ_i in the definition of p. Alice's private key is a list of exponents $(e_1, \dots, e_n) \in \mathbb{Z}^n$ where e_i shows how often each $\mathfrak{l}_i$ is used: Alice's public key is $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \star E_0 = E_A$, and if Bob's public key is E_B then the secret shared with Bob is $\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_n} \star E_B$. The key-exchange protocol works because $\star$ is a commutative group action: the ordering of the isogenies is not important.

The first constant-time CSIDH paper [136] specified each exponent e_i as being between 0 and a public constant m_i, and always computed m_i iterations of $\mathfrak{l}_i$, secretly discarding the dummy operations beyond e_i iterations. The original CSIDH paper [49] had allowed $e_i \in [-m_i, m_i]$; in the constant-time context this might seem to require m_i iterations of $\mathfrak{l}_i$ plus m_i iterations of $\mathfrak{l}_i^{-1}$, but [154] introduced a "2-point" algorithm with just m_i iterations, each iteration being only about 1/3 more expensive than before. All m_i were taken equal in [49], for example taking $e_i \in [-5, 5]$ for CSIDH-512; subsequent papers did better by allowing m_i to depend on i (as suggested in [49, Remark 14]) and accounting for the costs of $\mathfrak{l}_i$. Further speedups in the literature come from various techniques to speed up each $\mathfrak{l}_i$ computation and to merge work across sequences of $\mathfrak{l}_i$ computations.

4.2.1.1 Contributions of this chapter

This chapter introduces a new key space for CSIDH, and a new constant-time algorithm to evaluate the CSIDH group action. The new key space is not useful by itself—it slows down previous constant-time algorithms—and similarly the new constant-time algorithm is not useful for previous key spaces; but there is a synergy between the key space and the algorithm, and using both of them together produces a large improvement in the performance of constant-time CSIDH.

As a very small example of the new key space, assume that one is using just 6 primes and allows at most 6 isogeny computations, with each $\mathfrak{l}_i$ exponent being nonnegative. The standard key space chooses $(e_1, e_2, \ldots, e_6) \in \{0,1\}^6$, giving $2^6 = 64$ keys. The new key space, with 2 *batches* of 3 primes each, chooses $(e_1, e_2, \ldots, e_6) \in [0,3]^6$ with the condition that $e_1 + e_2 + e_3 \le 3$ and $e_4 + e_5 + e_6 \le 3$, giving $20^2 = 400$ keys. Similar comments apply when negative exponents are allowed.

The extreme case of putting each prime into a size-1 batch is not new: it is the standard key space. The opposite extreme, putting all primes into 1 giant batch, is also not new: putting a bound on the 1-norm of the key vector was highlighted in [146] as allowing the best tradeoffs between the number of isogenies and the size of the key space. In the above example, 1 giant batch of 6 primes gives 924 keys for 6 isogenies, i.e., 0.61 isogenies per key bit, compared to 1 isogeny per key bit for the standard key space.

However, plugging 1 giant batch into constant-time algorithms takes 36 isogenies for 924 keys, since each of the 6 primes uses 6 computations. Our intermediate example, 2 batches of 3 primes each, uses 18 isogenies for 400 keys, which is still many more isogenies per key bit than 6 isogenies for 64 keys.

We do better by evaluating isogenies differently. The central challenge tackled in this paper is to develop an efficient constant-time algorithm for the new key space, computing *any* isogeny in a batch using the *same* sequence of operations. This raises several questions:

1. How does one optimally compute a sequence of isogenies, and handle probabilistic failures in standard algorithms to compute $\mathfrak{l}_i$, while at the same time hiding which isogeny is computed? See Section 4.2.4 for the introduction of *atomic blocks* for these computations, and Section 4.2.5 for how to compute them in constant time.

2. How does one optimally set batches, compute private keys, and determine the number of isogenies per batch to match a required size of the key space? See Section 4.2.3 for analysis of the key space, and Section 4.2.6 for how to minimize the multiplication count.

3. How does one minimize the cycle count for constant-time software? Section 4.2.7 describes our software and low-level ideas; Section 4.2.8 presents the speeds.

Our constant-time algorithm combines several old and new techniques. For example, as observed in [23], Vélu's formulas have a Matryoshka-doll structure; we constrain the more recent $\sqrt{\text{é}}$lu formulas [19] in a way that creates a Matryoshka-doll structure. The cost of an isogeny computation exploiting this structure depends on the largest prime in the batch for the traditional formulas, but also on the smallest prime in the batch for $\sqrt{\text{é}}$lu. Standard algorithms to compute $\mathfrak{l}_i$ fail with probability $1/\ell_i$; to hide which ℓ_i in a batch is used we arrange for failures to occur with probability matching the smallest prime in the batch. Our batches consist of primes of similar sizes, to obtain the optimal tradeoffs between the cost per batch and the size of the key space. Further constant-time optimizations are described throughout the paper.

For comparability we report CSIDH-512 speeds, setting records in multiplications and in cycles for complete constant-time software. Partial analyses [23, 158, 35, 54] suggest that the post-quantum security level of CSIDH-512 is around 2^{60} qubit operations; for applications that want higher security levels, our software also supports larger sizes.

4.2.2 Background

This sections reviews the CSIDH group action, computations of individual ℓ-isogenies, strategies for computing sequences of isogenies, and previous constant-time algorithms.

4.2.2.1 The CSIDH group action

CSIDH [49] is a Diffie–Hellman-like key-exchange protocol based on isogenies of supersingular elliptic curves over a finite field $\mathbb{F}_p$. For a prime $p > 3$ an elliptic curve $E/\mathbb{F}_p$ is supersingular if and only if $\#E(\mathbb{F}_p) = p + 1$, where $E(\mathbb{F}_p)$ is its group of points over $\mathbb{F}_p$. CSIDH uses $p \equiv 3 \pmod 8$, and uses supersingular elliptic curves over $\mathbb{F}_p$ that can be written in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$ for $A \in \mathbb{F}_p \backslash \{-2, 2\}$. We call A the *Montgomery coefficient* of E_A. We write $\mathcal{E} = \{E_A : \#E_A(\mathbb{F}_p) = p+1\}$ for the set of CSIDH curves and $\mathcal{M} = \{A : E_A \in \mathcal{E}\}$ for the set of corresponding Montgomery coefficients. Curves E_A for distinct $A \in \mathcal{M}$ are non-isomorphic by [49, Proposition 8], and each $E_A(\mathbb{F}_p)$ is cyclic.

An isogeny is a nonzero map $\varphi : E \to E'$ which is given by rational functions and is compatible with elliptic-curve addition. An ℓ-isogeny is an isogeny of degree ℓ (as a rational map). Isogenies are typically defined by their kernels, i.e., by the points they map to ∞. Computing an ℓ-isogeny with Vélu's formulas requires a point P of order ℓ; the isogeny has kernel $\langle P \rangle$, and any point in $\langle P \rangle$ of order ℓ leads to the same isogeny.

The CSIDH prime p is chosen as $p = 4 \cdot \prod_{i=1}^{n} \ell_i - 1$ for small odd primes $\ell_1 < \ell_2 < \cdots < \ell_n$. If E_A is a curve in $\mathcal{E}$ then there are $\ell_j - 1$ points of order ℓ_j in $E_A(\mathbb{F}_p)$. Each of these points of order ℓ_j generates the kernel of an ℓ_j-isogeny $E_A \to E_{A'}$, which is the same isogeny for all of these points. The codomain $E_{A'}$ of this ℓ_j-isogeny is written $\mathfrak{l}_j \star E_A$.

Fix $i \in \mathbb{F}_{p^2}$ with $i^2 = -1$. Define $\tilde{E}_A(\mathbb{F}_p)$ as the set of points $(x, iy) \in E_A(\mathbb{F}_{p^2})$ where $x, y \in \mathbb{F}_p$, along with the neutral element; equivalently, $\tilde{E}_A(\mathbb{F}_p)$ is the image of $E_{-A}(\mathbb{F}_p)$ under the isomorphism $(x, y) \mapsto (-x, iy)$. For each ℓ_j, there are $\ell_j - 1$ points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$. Each of these points generates the kernel of an ℓ_j-isogeny $E_A \to E_{A''}$, the same isogeny for all of these points. The codomain $E_{A''}$ of this ℓ_j-isogeny is written $\mathfrak{l}_j^{-1} \star E_A$.

The isogeny from E_A to $\mathfrak{l}_j^{-1} \star E_A$ maps the points of order ℓ_j in $E_A(\mathbb{F}_p)$ to points of order ℓ_j, while mapping the points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$ to ∞ on $\mathfrak{l}_j^{-1} \star E_A$. The isogeny from E_A to $\mathfrak{l}_j \star E_A$ maps the points of order ℓ_j in $\tilde{E}_A(\mathbb{F}_p)$ to points of order ℓ_j, while mapping the points of order ℓ_j in $E_A(\mathbb{F}_p)$ to ∞ on $\mathfrak{l}_j \star E_A$.

Applying ℓ_i-isogenies induces a group action [23] of the commutative group $\mathbb{Z}^n$ on $\mathcal{E}$. An exponent vector $(e_1, \dots, e_n) \in \mathbb{Z}^n$ acts on the curve E_A to produce the curve $(\mathfrak{l}_1^{e_1} \dots \mathfrak{l}_n^{e_n}) \star E_A$, computed as a sequence having $|e_j|$ many ℓ_j-isogenies for each j, each isogeny using $\mathfrak{l}_j$ if $e_j > 0$ or $\mathfrak{l}_j^{-1}$ if $e_j < 0$.

The private key of each party is a (secret) vector $(e_1, \dots, e_n)$ sampled from a finite key space $\mathcal{K} \subset \mathbb{Z}^n$. To protect against meet-in-the-middle attacks, it is conventional to take $\#\mathcal{K} \geq 2^{2\lambda}$ for security 2^λ, but see [54] for arguments that smaller key spaces suffice. Beyond the size of $\mathcal{K}$, the specific choice of $\mathcal{K}$ has an important impact on efficiency.

Most previous CSIDH implementations have used one of two types of key space. Given an exponent bound vector $m = (m_1, \dots, m_n) \in \mathbb{Z}_{\geq 0}^n$, let $\mathcal{K}_m := \prod_{i=1}^n \{-m_i, \dots, m_i\}$ and $\mathcal{K}_m^+ := \prod_{i=1}^n \{0, \dots, m_i\}$. Clearly, $\#\mathcal{K}_m = \prod_{i=1}^n (2m_i + 1)$ and $\#\mathcal{K}_m^+ = \prod_{i=1}^n (m_i + 1)$. The original CSIDH paper [49] and [154] use $\mathcal{K}_m$ with $m = (5, \dots, 5)$ for CSIDH-512. It was suggested in [49, Remark 14] and shown in [56] that allowing the m_i to vary improves speed. The space $\mathcal{K}_m^+$ with $m = (10, \dots, 10)$ was used in [136] for CSIDH-512.

4.2.2.2 Computing isogenies

Let $P \in E_A$ be a point of order ℓ with x-coordinate in $\mathbb{F}_p$ and $\varphi : E_A \to E_{A'} = E_A/\langle P \rangle$ the ℓ-isogeny induced by P. The main computational task, called xISOG, is to compute (1) the Montgomery coefficient A' of the target curve $E_{A'}$ and (2) the images under φ of some specified points (normally zero, one, or two points) $Q \in E_A$ with x-coordinate in $\mathbb{F}_p$.

Vélu and $\sqrt{}$élu. The main algorithms for xISOG are Vélu's formulas [184] and $\sqrt{}$élu[16] [19]. The main computational task in both of these algorithms is to evaluate a polynomial

$$h_S(X) = \prod_{s \in S} (X - x([s]P)) \tag{4.1}$$

[16]Pronounced "square-root Vélu".

for some index set S. All the arithmetic is done using only x-coordinates.

Vélu's formulas evaluate $h_S(X)$ with $S = \{1, 2, 3, \dots, (\ell - 1)/2\}$ by first computing $x(P), x([2]P), \dots x([(\ell-1)/2]P)$ and then evaluating the product (4.1). This costs $O(\ell)$ field multiplications. Specifically, computing A' costs about 4ℓ field multiplications and computing the image of a point costs about 2ℓ extra field multiplications.

For $\sqrt{}$élu, in contrast to the linear algorithm of Vélu, the main part of the product is evaluated using a baby-step-giant-step strategy. It is simplest to evaluate $h_S(X)$ with $S = \{1, 3, 5, \dots, \ell-2\}$; this set S is split into a "box" $U \times V$ and leftover set W such that $S \leftrightarrow (U \times V) \cup W$. Then $h_S(X)$ is computed as the product of $h_W(X)$ with the resultant of $h_U(X)$ and a polynomial related to $h_V(X)$; see [19] for details. For each ℓ, one chooses U and V to minimize cost. Asymptotically, $\sqrt{}$élu uses $\tilde{O}(\sqrt{\ell})$ field multiplications.

One can view Vélu's formulas as a special case of $\sqrt{}$élu in which U and V are empty. This special case is optimal for small primes. The exact cutoff depends on lower-level algorithmic details but is around $\ell = 89$.

Sampling points of order ℓ. No efficient way is known to deterministically generate points of order ℓ in $E(\mathbb{F}_p)$. However, $E(\mathbb{F}_p)$ is cyclic of order $p+1$, so if T is a uniform random point then $P = [(p+1)/\ell]T$ will have order ℓ with probability $1 - 1/\ell$. This can—and often will—fail, and needs to be repeated until it succeeds. Once P has order ℓ, one can use P with Vélu or $\sqrt{}$élu.

Typically, one uses the Elligator 2 map [21] to sample points in $E(\mathbb{F}_p)$ or $\tilde{E}(\mathbb{F}_p)$. We discuss this approach in Appendix of this section.

4.2.2.3 Strategies

Computing the multiple $[(p+1)/\ell]T$ is very costly. If p has 512 bits then $(p+1)/\ell$ has almost 512 bits and this scalar multiplication costs thousands of field multiplications. The cost of scalar multiplication is typically amortized by pushing points through isogenies. This approach aims to compute a series of isogenies after only sampling one point on the initial curve.

Following [109], we call a method that computes a given series of isogenies a *strategy*. Informally, a strategy determines the order of isogeny evaluations, and how to obtain suitable kernel generators through either scalar multiplications or point evaluations. In the context of SIDH, *optimal strategies* (with the minimum computational cost) can be found [109]. When adapting this to CSIDH, there are three main complications: the choice of isogenies to be combined into a sequence, the possibility of point rejections due to wrong orders, and the pairwise different degrees of the involved isogenies. Indeed, [56] showed that a direct adaption of the method of [109] to CSIDH becomes infeasible when considering all possible permutations. Instead, several avenues for optimizing CSIDH strategies have been proposed, though none claims actual optimality.

Multiplicative strategy. A simple *multiplicative* strategy was used in the algorithm of [49]. Let D, a divisor of $p+1$, be the product of the degrees of the isogenies to be combined in one sequence. Sample a point T on the initial curve, and set $T \leftarrow [(p+1)/D]T$; now the order of T divides D. Compute $P \leftarrow [D/\ell]T$, where ℓ is the degree of the first isogeny. If $P = \infty$, skip this isogeny and continue with the next isogeny. If $P \neq \infty$, then P has order ℓ, so we can compute the required ℓ-isogeny φ, and push T through to get $T \leftarrow \varphi(T)$. Either way, the order of T now divides D/ℓ. For the next isogeny, say of degree ℓ', compute $P \leftarrow [D/(\ell\ell')]T$ as a potential kernel generator. Continue in the same fashion, pushing one point through each isogeny, until $T = \infty$. Note that the scalar multiplications reduce in length at each step: as observed in [137], processing the isogenies in decreasing degree order reduces the total cost.

SIMBA. In [49], the product D was taken as large as possible at each step. The SIMBA (Splitting Isogenies into Multiple Batches) strategy proposed in [136] limits D for better performance. SIMBA-M partitions the set of isogeny degrees $\{\ell_1, \ldots, \ell_n\}$ into M prides,[17] where the i-th pride contains all isogeny degrees ℓ_j for which $j \equiv i \mod M$. Each step, SIMBA picks as many isogenies as possible for a single pride, processing them as described above. This typically results in a smaller total degree D, making the initial scalar multiplication $T \leftarrow [(p+1)/D]T$ more expensive, while the later scalar multiplications of the form $P \leftarrow [D/\prod \ell_j]T$ become significantly cheaper. Overall, [136] reports that a significant speedup can be achieved for well-chosen values of M.

Point-pushing strategies. One can also push additional points through isogenies. For instance, when computing the first kernel generator as described above via $P \leftarrow [D/\ell]T$, one can save an intermediate point T' of small order divisible by ℓ', push both T and T' through the first isogeny, and then use T' to compute the next kernel generator via a smaller scalar multiplication. This may be more efficient than the multiplicative strategy, depending on the cost of evaluating φ at additional points compared to the savings due to cheaper scalar multiplications. However, except for very small isogeny degrees, the cost of evaluating additional points can be significantly higher than computing scalar multiplications. Thus, an optimal strategy is expected to be closer to the multiplicative strategy, only rarely pushing additional points through isogenies. In [56] optimal strategies are computed

[17]This refers to the term *pride of lions*. The term *batch* was used in [136]; we use *pride* here in order to distinguish between SIMBA batches and the batches of primes considered in CTIDH.

under the assumption of always choosing as many isogeny degrees as possible per sequence, and an increasing ordering of the involved degrees. It remains open whether other choices of primes per sequence, as in SIMBA, or different orderings of the degrees could yield a more optimized point-pushing strategy.

Remark 3. *The SIMBA approach is generalized in [104] with point-pushing strategies within prides, more efficient partitions of SIMBA prides, and their permutations. However, all optimization attempts have required imposing certain assumptions in order for the optimization problem to be solvable, and thus only produce conditionally optimal strategies. The comparison in [56] shows that all of these approaches give roughly the same performance results for constant-time CSIDH-512 algorithms: that is, within a margin of 4%.*

4.2.2.4 Previous constant-time algorithms

A private key $(e_1, \ldots, e_n)$ requires us to compute $|e_i|$ isogenies of degree ℓ_i (regardless of the strategy), so the running time of a naïve CSIDH algorithm depends directly on the key. Various constant-time approaches have been proposed to avoid this dependency.

What constant time means. A deterministic algorithm computes a function from inputs to outputs. A randomized algorithm is more complicated: it computes a function from inputs to distributions over outputs, since each run will, in general, depend on random bits generated inside the algorithm. Similarly, the time taken by an algorithm is a function from inputs to distributions of times. “Constant time” means that this function is constant: the distribution of algorithm time for input i matches the distribution of algorithm time for input i'. In other words, the algorithm time provides no information about the input.

In particular, if the input is a CSIDH curve and a private key, and the output is the result of the CSIDH action, then the algorithm time provides no information about the private key, and provides no information about the output.

Avoiding data flow from inputs to branches and array indices is sufficient to ensure the constant-time property for many definitions of algorithm time, and is the main focus of work on constant-time algorithms for CSIDH, including the work in this paper. Beware, however, that this is not sufficient if the definition changes to allow, e.g., a variable-time division instruction, like the division instructions on most computers.

The constant-time property also does not mean that time is deterministic. The paper [23] aims for time to be constant *and* deterministic, so as to be able to run in superposition on a quantum computer, but this costs extra and is not necessary for the objective of stopping timing attacks.

Structurally, every claim of constant-time *software* in the literature relies on various CPU instructions taking constant time, and could be undermined by CPU manufacturers adding timing variations to those instructions. The literature on constant-time software generally assumes, for example, that multiplication instructions take constant time, and declares that CPUs with variable-time multiplication instructions are out of scope. Formally, the constant-time claims are in a model of "time" where various instructions, including multiplications, take constant time.

Dummy isogenies. [136] used *dummy isogenies* to obtain a fixed number of isogenies per group action evaluation. Essentially, if e_i is sampled such that $|e_i| \leq m_i$, this amounts to computing m_i isogenies of degree ℓ_i, where $m_i - |e_i|$ of these are dummy computations whose results are simply discarded.[18] As noted in Section 4.2.2.2, this might require more than m_i attempts to sample a point of order ℓ_i, due to the point rejection probability of $1/\ell_i$. However, the number of attempts only depends on randomness and m_i, and is thus independent of the choice of e_i.

1-point and 2-point approaches. It is observed in [136] that if we compute multiple isogenies from a single sampled point, then the running time of the algorithm depends on the sign distribution of the private keys. Indeed, when a single point is sampled, only ℓ_i-isogenies with equal signs of the corresponding e_i can be combined in a strategy. Since this approach of combining isogenies is desirable for efficiency (see Section 4.2.2.3), [136] proposed eliminating this dependency by sampling e_i from $[0, 2m_i]$ instead of $[-m_i, m_i]$, although this requires computing twice as many isogenies per degree.

In order to mitigate this slowdown, [154] proposed sampling two points, $T_0 \in E_A(\mathbb{F}_p)$ and $T_1 \in \tilde{E}_A(\mathbb{F}_p)$. For each isogeny in the sequence, one picks the kernel generator according to the sign of the corresponding e_i. This approach combines isogeny computations independent of key signs, and thus goes back to sampling e_i from $[-m_i, m_i]$ at the cost of pushing two points through each isogeny instead of one.

A dummy-free variant of the 2-point approach was proposed in [51]. This requires roughly twice as many isogenies, but may be useful in situations where fault-injection attacks play an important role. We return to this approach in Appendix.

[18]In [136] some other computations are performed inside dummy isogenies, which facilitate later steps in the algorithm. We omit the details here, since we only use simple dummy isogenies as described above.

4.2.3 Batching and key spaces

The main conceptual novelty in CTIDH is the organization of primes and isogenies in batches. For this we define a new batch-oriented key space, which is slightly more complicated than the key spaces $\mathcal{K}_m$ and $\mathcal{K}_m^+$ mentioned in Section 4.2.2.

Batching primes. In CTIDH, the sequence of primes $(\ell_1, \dots, \ell_n)$ is partitioned into a series of *batches*: subsequences of consecutive primes. Let $0 < B \leq n$ be the number of batches; we represent the sequence of the batch sizes by a vector $N = (N_1, \dots, N_B) \in \mathbb{Z}_{>0}^B$ with $\sum_{i=1}^B N_i = n$. We relabel the primes in batches as: $(\ell_{1,1}, \dots, \ell_{1,N_1}) := (\ell_1, \dots, \ell_{N_1})$, $(\ell_{2,1}, \dots, \ell_{2,N_2}) := (\ell_{N_1+1}, \dots, \ell_{N_1+N_2}), \dots, (\ell_{B,1}, \dots, \ell_{B,N_B}) := (\ell_{n-N_B+1}, \dots, \ell_n)$. If $\ell_{i,j}$ corresponds to ℓ_k, then we also write $\mathfrak{l}_{i,j}$ for $\mathfrak{l}_k$ and $e_{i,j}$ for e_k.

Example 1: Say we have $n = 6$ primes, $(\ell_1, \dots, \ell_6)$. If we take $B = 3$ and $N = (3, 2, 1)$, then $(\ell_{1,1}, \ell_{1,2}, \ell_{1,3}) = (\ell_1, \ell_2, \ell_3)$, $(\ell_{2,1}, \ell_{2,2}) = (\ell_4, \ell_5)$, and $(\ell_{3,1}) = (\ell_6)$.

Batching isogenies. Consider the i-th batch of primes $(\ell_{i,1}, \dots, \ell_{i,N_i})$. Rather than setting a bound $m_{i,j} \geq |e_{i,j}|$ for the number of $\ell_{i,j}$-isogenies for each $1 \leq j \leq N_i$, we set a bound $m_i \geq \sum_{j=1}^{N_i} |e_{i,j}|$ and compute m_i isogenies from the batch $(\ell_{i,1}, \dots, \ell_{i,N_i})$. This looks analogous to the use of dummy operations in the previous constant-time algorithms, but it gives a larger keyspace per isogeny computed because of the ambiguity between the degrees in a batch. Moreover, we will show that it is possible to evaluate any isogeny within one batch in the same constant time.

Extreme batching choices correspond to well-known approaches to the group action evaluation: one prime per batch ($B = n$ and $N = (1, \dots, 1)$) was considered in [49]; one n-prime batch ($B = 1$ and $N = (n)$) is considered in [23] for the quantum oracle evaluation and in [146] as a speedup for CSIDH. The intermediate cases are new, and, as we will show, faster.

The new key space. For $N \in \mathbb{Z}_{>0}^B$ and $m \in \mathbb{Z}_{\geq 0}^B$, we define

$$\mathcal{K}_{N,m} := \left\{ (e_1, \dots, e_n) \in \mathbb{Z}^n \mid \textstyle\sum_{j=1}^{N_i} |e_{i,j}| \leq m_i \text{ for } 1 \leq i \leq B \right\}.$$

We may see $\mathcal{K}_{N,m}$ as a generalization of $\mathcal{K}_m$.

Lemma 1. *We have*

$$\#\mathcal{K}_{N,m} = \prod_{i=1}^{B} \Phi(N_i, m_i), \quad \textit{where} \quad \Phi(x, y) = \sum_{k=0}^{\min\{x,y\}} \binom{x}{k} 2^k \binom{y}{k}$$

counts the vectors in $\mathbb{Z}^x$ with 1-norm at most y.

Proof. The size of the key space is the product of the sizes for each batch. In $\Phi(x, y)$ the number of nonzero entries in the x positions is k and there are $\binom{x}{k}$ ways to determine which entries are nonzero. For each of the nonzero entries there are 2 ways to choose the sign. The vector of partial sums over these k nonzero entries has k different integers in $[1, y]$ and each vector uniquely matches one assignment of partial sums. There are $\binom{y}{k}$ ways to pick k different integers in $[1, y]$. □

4.2.4 Isogeny atomic blocks

In this section we formalize the concept of *isogeny atomic blocks* (ABs), subroutines that have been widely used in constant-time CSIDH algorithms but never formalized before. The first step of an algorithm chooses a series of degrees for which isogenies still need to be computed, and then uses, for example, the multiplicative strategy (Section 4.2.2.3) to compute a sequence of isogenies of those degrees. The next step chooses a possibly different series of degrees, and computes another sequence of isogenies. Each step of the computation is the evaluation of not one isogeny, but a sequence of isogenies. Atomic blocks formalize these steps.

Square-free ABs generalize the approach we take when evaluating the CSIDH group action with the traditional key spaces $\mathcal{K}_m$ and $\mathcal{K}_m^+$ as in Algorithm 7. *Restricted square-free ABs* are used to evaluate the group action using the batching idea with keys in $\mathcal{K}_{N,m}$; with details in Algorithm 8. We postpone the explicit construction of ABs to Section 4.2.5.

4.2.4.1 Square-free atomic blocks

Definition 19 (Square-free ABs)**.** *Let $R \subseteq \{-1, 0, 1\}$ and $I = (I_1, \ldots, I_k) \in \mathbb{Z}^k$ such that $1 \leq I_1 < I_2 < \cdots < I_k \leq n$. A* square-free atomic block *of length k is a probabilistic algorithm $\alpha_{R,I}$ taking inputs $A \in \mathcal{M}$ and $\epsilon \in R^k$ and returning $A' \in \mathcal{M}$ and $f \in \{0, 1\}^k$ such that $E_{A'} = (\prod_i \mathfrak{l}_{I_i}^{f_i \cdot \epsilon_i}) \star E_A$, satisfying the following two properties:*

1. *there is a function σ such that, for each (A, ϵ), the distribution of f, given that (A', f) is returned by $\alpha_{R,I}$ on input (A, ϵ), is $\sigma(R, I)$, and*
2. *there is a function τ such that, for each (A, ϵ) and each f, the distribution of the time taken by $\alpha_{R,I}$, given that (A', f) is returned by $\alpha_{R,I}$ on input (A, ϵ), is $\tau(R, I, f)$.*

Suppose an algorithm evaluates the group action on input $e \in \mathcal{K}$ and $A \in \mathcal{M}$ using a sequence of square-free AB calls $(A', f) \leftarrow \alpha_{R,I}(A, \epsilon)$. If in each step the choice of R and I are independent of e, the algorithm does not leak information about e through timing.

This is illustrated by Algorithm 7, which expresses the constant-time group action from [154] using a sequence of square-free ABs with $R =$

Algorithm 7: Generalization of [154, Algorithm 3], replacing the inner loop with any square-free AB with $R = \{-1, 0, 1\}$. Keys are in $\mathcal{K}_m$.

```
  Parameters: $m = (m_1, \dots, m_n)$
  Input: $A \in \mathcal{M}$, $e = (e_1, \dots, e_n) \in \mathcal{K}_m$
  Output: $A'$ with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$
1 $(\mu_1, \dots, \mu_n) \leftarrow (m_1, \dots, m_n)$ ;
2 while $(\mu_1, \dots, \mu_n) \neq (0, \dots, 0)$ do
3     Let $I = (I_1, \dots, I_k)$ s.t. $I_1 < \dots < I_k$ and
        $\{I_1, \dots, I_k\} = \{1 \le i \le n \mid \mu_i > 0\}$ ;
4     for $1 \le i \le k$ do
5         $\epsilon_i \leftarrow$ Sign($e_{I_i}$) ;   // 1 if $e_{I_i} > 0$; 0 if $e_{I_i} = 0$; -1 if $e_{I_i} < 0$
6     $(A, f) \leftarrow \alpha_{R,I}(A, (\epsilon_1, \dots, \epsilon_k))$ ;   // Square-free AB
7     for $1 \le i \le k$ do
8         $(\mu_{I_i}, e_{I_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i} - \epsilon_i \cdot f_i)$ ;
9 return $A$
```

$\{-1, 0, 1\}$ to evaluate the action for keys in $\mathcal{K}_m$. The choices of R and I are independent of e for each AB $\alpha_{R,I}$, and all other steps can be easily made constant-time. The choice of I in Step 3 may vary between different executions, due to the varying failure vectors f of previously evaluated ABs. However this only depends on the initial choice of m_i, and is independent of e.

Remark 4. *The constant-time group action from [136] can also be expressed simply in terms of ABs. The algorithm is extremely similar to Algorithm 7, using $\mathcal{K}_m^+$ in place of $\mathcal{K}_m$ (the algorithm of [136] uses $m = (10, \dots, 10)$) and $R = \{0, 1\}$ in place of $\{-1, 0, 1\}$. Line 5 can be simplified to $\epsilon_i \leftarrow 1$ if $e_{I_i} \neq 0$, or 0 if $e_{I_i} = 0$.*

Remark 5. *The distribution of f depends on how the ABs are constructed. In [136] and [154], $Pr(f_i = 0) = 1/\ell_{I_i}$ for all i. In [54], f is always $(1, 1, \dots, 1)$.*

4.2.4.2 Restricted square-free atomic blocks

In the language of Section 4.2.3, restricted square-free ABs are generalizations of square-free ABs that further do not leak information on which of the primes we have chosen from a batch.

Definition 20 (Restricted square-free ABs). *Let $R \subseteq \{-1, 0, 1\}$, $B \ge 1$, and $I = (I_1, \dots, I_k) \in \mathbb{Z}^k$ such that $1 \le I_1 < I_2 < \dots < I_k \le B$. A* restricted square-free atomic block *of length k is a probabilistic algorithm $\beta_{R,I}$ taking inputs $A \in \mathcal{M}$, $\epsilon \in R^k$, and $J \in \mathbb{Z}^k$ with $1 \le J_i \le N_{I_i}$ for all $1 \le i \le k$, and returning $A' \in \mathcal{M}$ and $f \in \{0,1\}^k$ such that $E_{A'} = (\prod_i \mathfrak{l}_{I_i,J_i}^{f_i \cdot \epsilon_i}) \star E_A$, satisfying the following two properties:*

1. *there is a function σ such that, for each (A, ϵ, J), the distribution of f, given that (A', f) is returned by $\beta_{R,I}$ on input (A, ϵ, J), is $\sigma(R, I)$; and*

2. *there is a function τ such that, for each (A, ϵ, J) and each f, the distribution of the time taken by $\beta_{R,I}$, given that (A', f) is returned by $\beta_{R,I}$ on input (A, ϵ, J), is $\tau(R, I, f)$.*

Algorithm 8 uses restricted square-free ABs with $R = \{-1, 0, 1\}$ to compute group actions for keys in $\mathcal{K}_{N,m}$; it may be considered a generalization of Algorithm 7.

Algorithm 8: A constant-time group action for keys in $\mathcal{K}_{N,m}$ based on restricted square-free ABs with $R = \{-1, 0, 1\}$.

```
   Parameters: $N$, $m$, $B$
   Input: $A \in \mathcal{M}$, $e = (e_1, \dots, e_n) \in \mathcal{K}_{N,m}$
   Output: $A'$ with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$
1  $(\mu_1, \dots, \mu_B) \leftarrow (m_1, \dots, m_B)$ ;
2  while $(\mu_1, \dots, \mu_B) \neq (0, \dots, 0)$ do
3      Let $I = (I_1, \dots, I_k)$ s.t. $I_1 < \cdots < I_k$ and
        $\{I_1, \dots, I_k\} = \{1 \le i \le B \mid \mu_i > 0\}$ ;
4      for $1 \le i \le k$ do
5          if there exists $j$ such that $e_{I_i,j} \neq 0$ then
6              $J_i \leftarrow$ some such $j$
7          else
8              $J_i \leftarrow$ any element of $\{1, \dots, N_{I_i}\}$
9          $\epsilon_i \leftarrow \mathtt{Sign}(e_{I_i,J_i})$ ;   // 1 if $e_{I_i,J_i} > 0$; 0 if $e_{I_i,J_i} = 0$; -1 if
            $e_{I_i,J_i} < 0$
10     $(A, f) \leftarrow \beta_{R,I}(A, (\epsilon_1, \dots, \epsilon_k), J)$ ;  // Restricted square-free AB
11     for $1 \le i \le k$ do
12         $(\mu_{I_i}, e_{I_i,J_i}) \leftarrow (\mu_{I_i} - f_i, e_{I_i,J_i} - \epsilon_i \cdot f_i)$ ;
13 return $A$
```

4.2.5 Evaluating atomic blocks in constant time

This section introduces the algorithm used in CTIDH to realize the restricted square-free atomic block $\beta_{R,I}$ introduced in Section 4.2.4. Throughout this section, R is $\{-1,0,1\}$.

As a warmup, Section 4.2.5.1 recasts the inner loop of [154, Algorithm 3] as a realization of the square-free atomic block $\alpha_{R,I}$. We first present the algorithm in a simpler variable-time form (Algorithm 9) and then explain the small changes needed to eliminate timing leaks, obtaining $\alpha_{R,I}$.

Section 4.2.5.3 presents our new algorithm to realize $\beta_{R,I}$. The extra challenge here is to hide which prime is being used within each batch. Again we begin by presenting a simpler variable-time algorithm (Algorithm 10) and then explain how to eliminate timing leaks.

4.2.5.1 Square-free atomic blocks for isogeny evaluation

Algorithm 9 translates the inner loop of [154, Algorithm 3] to the AB framework. The inputs are $A \in \mathcal{M}$ and $\epsilon \in \{-1,0,1\}^k$. The goal is to compute k isogenies of degrees $\ell_{I_1},\dots,\ell_{I_k}$, but some of these computations may fail. The outputs are a vector $f \in \{0,1\}^k$ recording which of the computations succeeded, and A' such that $(\prod_i \mathfrak{l}_{I_i}^{f_i \cdot \epsilon_i}) \star E_A = E_{A'}$.

The algorithm uses the 2-point approach with dummy isogenies. It uses two subroutines:

- `UniformRandomPoints` takes $A \in \mathcal{M}$, and returns a uniform random pair of points (T_0, T_1), with $T_0 \in E_A(\mathbb{F}_p)$ and $T_1 \in \tilde{E}_A(\mathbb{F}_p)$; i.e., T_0 is a uniform random element of $E_A(\mathbb{F}_p)$, and T_1, independent of T_0, is a uniform random element of $\tilde{E}_A(\mathbb{F}_p)$.
- `Isogeny` takes $A \in \mathcal{M}$, a point P in $E_A(\mathbb{F}_{p^2})$ with x-coordinate in $\mathbb{F}_p$ generating the kernel of an ℓ_{I_j}-isogeny $\varphi : E_A \to E_{A'} = E_A/\langle P\rangle$, and a tuple of points $(Q_1,\dots,Q_t)$, and returns A' and $(\varphi(Q_1),\dots,\varphi(Q_t))$.

See Appendix of this chapter for analysis of the Elligator alternative to `UniformRandomPoints`.

Remark 6. *Algorithm 9 uses a multiplicative strategy, but it can easily be modified to use a SIMBA or point-pushing strategy, which is much more efficient in general [154, 56]. The isogeny algorithm can be Vélu or* $\sqrt{}$élu, *whichever is more efficient for the given degree.*

4.2.5.2 Modifying Algorithm 9 to eliminate timing leaks

The following standard modifications to Algorithm 9 produce an algorithm meeting Definition 19, the definition of a square-free atomic block.

Observe first that $f_j = 1$ if and only if the prime ℓ_{I_j} divides the order of the current T_s. This is equivalent to ℓ_{I_j} dividing the order of the initially

Algorithm 9: Inner loop of [154, Algorithm 3].

```
   Parameters: $k \in \mathbb{Z}, R = \{-1,0,1\}, I \in \mathbb{Z}_{\geq 0}^k$
   Input: $A \in \mathcal{M}, \epsilon \in \{-1,0,1\}^k$
   Output: $A' \in \mathcal{M}, f \in \{0,1\}^k$
1  $(T_0, T_1) \leftarrow$ UniformRandomPoints($A$) ;
2  $(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ where $r = 4\prod_{i \notin I} \ell_i$ ;
3  $r' \leftarrow \prod_{i \in I} \ell_i$ ;
4  for $j = k$ down to 1 do
5      $r' \leftarrow r'/\ell_{I_j}$ ;
6      $s \leftarrow$ SignBit($\epsilon_j$) ;                  // 1 if $\epsilon_j < 0$, otherwise 0
7      $P \leftarrow [r']T_s$ ;
8      if $P \neq \infty$ then                // branch without secret information
9          $f_j \leftarrow 1$ ;
10         $(A', (T'_0, T'_1)) \leftarrow$ Isogeny($A$, $P$, $(T_0, T_1)$, $I_j$) ;
11         if $\epsilon_j \neq 0$ then          // branch with secret information
12             $(A, T_0, T_1) \leftarrow (A', T'_0, T'_1)$
13         else
14             $T_s \leftarrow [\ell_{I_j}]T_s$ ;
15     else
16         $f_j \leftarrow 0$ ;
17     $T_{1-s} \leftarrow [\ell_{I_j}]T_{1-s}$ ;
18 return $A$, $f$
```

sampled point T_s (since T_s has been modified only by multiplication by scalars that are not divisible by ℓ_{I_j}, and by isogenies of degrees not divisible by ℓ_{I_j}). This has probability $1-1/\ell_{I_j}$, since the initial T_s is a uniform random point in a cyclic group of size $p+1$. These probabilities are independent across j, since (T_0, T_1) is a uniform random pair of points.

To summarize, the distribution of the f vector has position j set with probability $1 - 1/\ell_{I_j}$, independently across j. This distribution is a function purely of I, independent of (A, ϵ), as required. What follows are algorithm modifications to ensure that the time distribution is a function purely of (I, f); these modifications do not affect f.

Step 7, taking T_0 if $s = 0$ or T_1 if $s = 1$, is replaced with a constant-time point selection: e.g., taking the bitwise XOR $T_0 \oplus T_1$, then ANDing each bit with s, and then XORing the result with T_0. Similar comments apply to the subsequent uses of T_s and T_{1-s}. It is simplest to merge all of these selections into a constant-time swap of T_0, T_1 when $s = 1$, followed by a constant-time swap back at the bottom of the loop. The adjacent swaps at the bottom of one loop and the top of the next loop can be merged, analogous merging is standard in constant-time versions of the Montgomery ladder for scalar multiplication.

Step 11 determines whether an actual isogeny or a dummy isogeny has to be computed. The conditional assignment to (A, T_0, T_1) in the first case is replaced with unconditional constant-time point selection. The conditional operation in the second case is replaced with an unconditional operation, multiplying T_s by ℓ_{I_j} in both cases. This changes the point T_s in the first case, but does not change the order of T_s (since the isogeny has already removed ℓ_{I_j} from the order of T_s in the first case), and all that matters for the algorithm is the order. See [136, 154] for a slightly more efficient approach, merging the multiplication by ℓ_{I_j} into a dummy isogeny computation.

The branch in Step 8 is determined by public information f_j and does not need to be modified. The isogeny computation inside `Isogeny` takes constant time with standard algorithms; at a lower level, arithmetic in $\mathbb{F}_p$ is handled by constant-time subroutines, not by subroutines that try to save time by suppressing leading zero bits. The computation of `UniformRandomPoints` takes variable time with standard algorithms, but the time distribution is independent of the curve provided as input.

The total time is the sum for initialization (`UniformRandomPoints`, computation of r and r', initial scalar multiplication), f_j computation (division, scalar multiplications, selection), and computations when $f_j = 1$ (`Isogeny`, scalar multiplication, selection). This sum is a function purely of (I, f), independent of (A, ϵ), as required.

4.2.5.3 Restricted square-free atomic blocks

We now consider the more difficult goal of hiding which isogeny is being computed within each batch. We present first the high-level algorithm (Algorithm 10), then the `PointAccept` and `MatryoshkaIsogeny` subroutines, and finally the algorithm modifications to meet Definition 20.

The inputs to Algorithm 10 are $A \in \mathcal{M}$, $\epsilon \in \{-1, 0, 1\}^k$, and $J \in \mathbb{Z}^k$. The goal is to compute k isogenies of degrees $\ell_{I_1,J_1}, \dots, \ell_{I_k,J_k}$. The outputs are $A' \in \mathcal{M}$ and $f \in \{0,1\}^k$ such that $(\prod_i \mathfrak{l}_{I_i,J_i}^{f_i \cdot \epsilon_i}) \star E_A = E_{A'}$.

Like Algorithm 9, Algorithm 10 uses a 2-point approach and dummy isogenies. It uses the following subroutines:

- `UniformRandomPoints` is as before.
- `PointAccept` replaces the check $P \neq \infty$ to prevent timing leakage. It takes a point P and $I_j, J_j \in \mathbb{Z}$ such that P either has order ℓ_{I_j,J_j} or 1, and outputs either 0 or 1, under the condition that the output is 0 whenever $P = \infty$.
- `MatryoshkaIsogeny` replaces `Isogeny` from Algorithm 9. There is an extra input J_j indicating the secret position within a batch.

Algorithm 10: The CTIDH inner loop.

```
   Parameters: $k \in \mathbb{Z}, R = \{-1,0,1\}, I \in \mathbb{Z}^k_{\geq 0}$
   Input: $A \in \mathcal{M}, \epsilon \in \{-1,0,1\}^k, J \in \mathbb{Z}^k_{>0}$
   Output: $A' \in \mathcal{M}, f \in \{0,1\}^k$
1  $(T_0, T_1) \leftarrow$ UniformRandomPoints($A$) ;
2  $(T_0, T_1) \leftarrow ([r]T_0, [r]T_1)$ where $r = 4\prod_{i \notin I}\prod_{1 \leq j \leq N_i} \ell_{i,j}$ ;
3  $(T_0, T_1) \leftarrow ([\tilde{r}]T_0, [\tilde{r}]T_1)$ where $\tilde{r} = \prod_{i \in I}\prod_{1 \leq j \leq N_i, j \neq J_i} \ell_{i,j}$ ;   // hide
    selection
4  $r' \leftarrow \prod_{i \in I} \ell_{i,J_i}$ ;                                  // hide selection
5  for $j = k$ down to $1$ do
6      $r' \leftarrow r'/\ell_{I_j,J_j}$ ;                    // hide $\ell_{I_j,J_j}$, batch is public
7      $s \leftarrow$ SignBit($\epsilon_j$) ;                       // 1 if $\epsilon_j < 0$, otherwise 0
8      $P \leftarrow [r']T_s$ ;                          // hide $\ell_{I_j,J_j}$, batch is public
9      $f_j \leftarrow$ PointAccept($P$, $I_j$, $J_j$) ;
10     if $f_j = 1$ then          // this branch is on public information
11         $(A', (T'_0, T'_1)) \leftarrow$ MatryoshkaIsogeny($A$, $P$, $(T_0, T_1), I_j, J_j$) ;
12         if $\epsilon_j \neq 0$ then            // branch with secret information
13             $(A, T_0, T_1) \leftarrow (A', T'_0, T'_1)$
14     $(T_0, T_1) \leftarrow ([\ell_{I_j,J_j}]T_0, [\ell_{I_j,J_j}]T_1)$ ;            // hide selection
15 return $A$, $f$
```

Note that the output of PointAccept can be 0 when $P \neq \infty$, so we add a multiplication by ℓ_{I_j,J_j} in Step 14 to make sure we continue the loop with points of expected order.

4.2.5.4 PointAccept

Step 8 of Algorithm 10 computes a potential kernel generator P. The probability that $P = \infty$ is $1/\ell_{I_j,J_j}$, which depends on J_j. For the batch $(\ell_{I_j,1}, \ldots, \ell_{I_j,N_{I_j}})$, PointAccept artificially increases this probability to $1/\ell_{I_j,1}$, independent of ℓ_{I_j,J_j}, by tossing a coin with success probability

$$\gamma = \frac{\ell_{I_j,J_j} \cdot (\ell_{I_j,1} - 1)}{\ell_{I_j,1} \cdot (\ell_{I_j,J_j} - 1)}$$

and only returning $f_j = 1$ if $P \neq \infty$ *and* the coin toss is successful. The probability that the output is 1 is then $\gamma \cdot (1 - 1/\ell_{I_j,J_j}) = 1 - 1/\ell_{I_j,1}$, which is independent of J_j. Thus the batch can fail publicly.

4.2.5.5 MatryoshkaIsogeny

`MatryoshkaIsogeny` replaces the `Isogeny` computation. It takes the Montgomery coefficient of a curve E_A, a batch $(\ell_{i,1}, \ldots, \ell_{i,N_i})$, an isogeny index j within the batch, a point P of order $\ell_{i,j}$ generating the kernel of an isogeny $\varphi : E_A \to E_A/\langle P\rangle = E_{A'}$, and a tuple of points $(Q_1, \ldots, Q_t)$, and returns A' and $(\varphi(Q_1), \ldots, \varphi(Q_t))$. `MatryoshkaIsogeny` is computed with cost independent of j.

For Vélu's formulas, [23] showed how to compute any ℓ_i-isogeny for $\ell_i \le \ell$ using the computation of an ℓ-isogeny and masking. The first step of computing (4.1) is to compute $x(P), x([2]P), \ldots, x([(\ell-1)/2]P)$. This includes the computation for smaller ℓ_i; [23] described this as a Matryoshka-doll property.

In this chapter we specialize the $\surd$élu formulas so as to obtain a Matryoshka-doll structure. We define the sets U and V, introduced in Section 4.2.2.2, as the optimal choices for the *smallest* degree in the batch: i.e., $\ell_{i,1}$. The leftover set W is chosen to make the formulas work even for the *largest* prime ℓ_{i,N_i} in the batch. Then the baby-step giant-step algorithm stays unchanged; while we iterate through W we save the intermediate results corresponding to all degrees $\ell_{i,j}$ in the batch. In the final step, we select the result corresponding to the index j that we wanted to compute.

The sets U and V have size around $\sqrt{\ell_{i,1}}$. If the primes in the batch are sufficiently close then the rounded values match or are marginally different, meaning that the Matryoshka-like formulas are at worst marginally slower than the optimal formulas for ℓ_{i,N_i}.

4.2.5.6 Modifying Algorithm 10 to eliminate timing leaks

We now indicate algorithm modifications to meet Definition 20, the definition of a restricted square-free atomic block.

As in Section 4.2.5.2, we begin with the distribution of f. For each input (A, ϵ, J), the distribution has f_j set with probability $1 - 1/\ell_{I_j,1}$ (not $1 - 1/\ell_{I_j,J_j}$; see Section 4.2.5.4), independently across j. This distribution is a function purely of I, independent of (A, ϵ, J), as required. What remains is to ensure that the time distribution is a function purely of (I, f).

There are secret scalars $\tilde{r}$, r', and ℓ_{I_j,J_j} used in various scalar multiplications in Steps 3, 8, and 14. Standard integer-arithmetic algorithms that dynamically suppress leading zero bits are replaced by constant-time algorithms that always use the maximum number of bits, and variable-time scalar-multiplication algorithms are replaced by a constant-time Montgomery ladder, as in [23]. It is straightforward to compute an upper bound on each scalar in Algorithm 10. See Section 4.2.7 for faster alternatives.

Section 4.2.5.5 explains how to compute `MatryoshkaIsogeny` in time that depends only on the batch, not on the selection of a prime within the batch. Everything else is as in Section 4.2.5.2: the distribution of `UniformRan-`

`domPoints` timings is independent of the inputs, Step 8 uses constant-time selection, the branch in Step 12 is replaced by constant-time selection, and the branch in Step 10 does not need to be modified.

4.2.6 Strategies and parameters for CTIDH

The optimization process for previous constant-time algorithms for CSIDH has two levels. The bottom level tries to minimize the cost of each AB, for example by optimizing $\sqrt{}$élu parameters and searching for a choice of strategy from Section 4.2.2.3. The top level searches for a choice of exponent bounds $m = (m_1, \dots, m_n)$, trying to minimize the total AB cost subject to the key space reaching a specified size. A cost function that models the cost of an AB, taking account of the bottom-level search, is plugged into the top-level search.

Optimizing CTIDH is more complicated. There is a new top level, searching for a choice of batch sizes $N = (N_1, \dots, N_B)$. These batch sizes influence the success chance and cost of an AB at the bottom level: see Sections 4.2.5.4 and 4.2.5.5. They also influence the total cost of any particular choice of 1-norm bounds $m = (m_1, \dots, m_B)$ at the middle level. The size of the key space depends on both N and m; see Lemma 1.

This section describes a reasonably efficient method to search for CTIDH parameters.

Strategies for CTIDH. We save time at the lowest level of the search by simply using multiplicative strategies. As in previous papers, it would be easy to adapt Algorithm 10 to use SIMBA or optimized point-pushing strategies or both, giving many further parameters that could be explored with more search time, but this is unlikely to produce large benefits.

Seen from a high level, evaluating ABs multiplicatively in CTIDH has a similar effect to SIMBA strategies for previous algorithms. For example, SIMBA-N_1 for traditional batch sizes $(1, \dots, 1)$ limits each AB to at most n/N_1 isogenies (if n is divisible by N_1), in order to save multiplicative effort. Now consider CTIDH where all B batches have size N_1, i.e., $N = (N_1, \dots, N_1)$. Each CTIDH AB then computes at most $B = n/N_1$ isogenies, saving multiplicative effort in the same way.

One could split a CTIDH AB into further SIMBA prides, but [136] already shows that most of the benefit of SIMBA comes from putting some cap on the number of isogenies in an AB; the exact choice of cap is relatively unimportant. One could also try to optimize point-pushing strategies as an alternative to multiplicative strategies, as an alternative to SIMBA, or within each SIMBA pride, but the searches in [104] and [56] suggest that optimizing these strategies saves at most a small percentage in the number of multiplications, while incurring overhead for managing additional points.

Cost functions for CTIDH. The search through various CTIDH batching configuration vectors N and 1-norm bound vectors m tries to minimize a function $C(N,m)$, a model of the cost of a group-action evaluation. The numerical search examples later in this section use the following cost function: the average number of multiplications (counting squarings as multiplications) used by the CTIDH algorithms, including the speedups described in Section 4.2.7.

One way to compute this function is to statistically approximate it: run the software from Section 4.2.7 many times, inspect the multiplication counter built into the software, and take the average over many experiments. A more efficient way to compute the same function with the same accuracy is with a simulator that skips the multiplications but still counts how many there are. Our simulator, despite being written in Python, is about 50 times faster than the software from Section 4.2.7.

However, using a statistical approximation raises concerns about the impact of statistical variations. So, instead of using the software or the simulator, we directly compute the average cost of the first AB, the average cost of the second AB, etc., stopping when the probability of needing any further AB is below 10^{-9}.

Batch b, with smallest prime $\ell_{b,1}$, has success probability $1-1/\ell_{b,1}$ from each AB, so the chance q_b of reaching m_b successes within R ABs is the sum of the coefficients of $x^{m_b}, x^{m_b+1}, \ldots$ in the polynomial $(1/\ell_{b,1}+(1-1/\ell_{b,1})x)^R$. Batches are independent, so $q_1 q_2 \cdots q_B$ is the probability of not needing any further AB. Note that multiplying the polynomial $(1/\ell_{b,1}+(1-1/\ell_{b,1})x)^R$ by $1/\ell_{b,1}+(1-1/\ell_{b,1})x$ for each increase in R is more efficient than computing binomial coefficients.

Computing the cost of an AB (times the probability that the AB occurs) is more complicated. Splitting the analysis into 2^B cases—e.g., one case, occurring with probability $(1-q_1)(1-q_2)\cdots(1-q_B)$, is that all B batches still remain to be done—might be workable, since B is not very large and one can skip cases that occur with very low probability. We instead take the following approach. Fix b. The probability that batch b is in the AB is $1-q_b$; the probability that batch a is in the AB for exactly j values $a < b$ is the coefficient of x^j in the polynomial $\prod_{a<b}(q_a + (1-q_a)x)$; and the probability that batch c is in the AB for exactly k values $c > b$ is the coefficient of x^k in the polynomial $\prod_{c>b}(q_c + (1-q_c)x)$. There are $O(B^2)$ possibilities for (j,k); each possibility determines the total number of batches in the AB and the position of b in the AB, assuming b is in the AB. For the AB algorithms considered here, this is enough information to determine the contribution of batch b to the cost of the AB. Our Python implementation of this approach has similar cost to 100 runs of the simulator, depending on B.

We also explored various simpler possibilities for cost functions. A deterministic model of ABs is easier to compute and simulates real costs reasonably well, leading to parameters whose observed costs were consistently within 10% of the best costs we found via the cost function defined above.

Optimizing the 1-norm bounds. Given a fixed configuration N of B batches, we use a greedy algorithm as in [56] to search for a 1-norm bound vector m as follows:

1. Choose an initial $m = (m_1, \dots, m_B)$ such that $\mathcal{K}_{N,m}$ is large enough, and set $C_{\min} \leftarrow C(N, m)$.
2. For each i in $\{1, \dots, B\}$, do the following:
    (a) Set $\tilde{m} \leftarrow (m_1, \dots, m_{i-1}, m_i - 1, m_{i+1}, \dots, m_B)$.
    (b) If $\mathcal{K}_{N,\tilde{m}}$ is large enough, set $(m, C_{\min}) \leftarrow (\tilde{m}, C(N, \tilde{m}))$.
    (c) Else, set $\tilde{m}' \leftarrow \tilde{m}$, and for each $j \neq i$ in $\{1, \dots, B\}$ do the following:
        i. Set $\tilde{m} \leftarrow (\tilde{m}'_1, \dots, \tilde{m}'_{j-1}, \tilde{m}'_j + 1, \tilde{m}'_{j+1}, \dots, \tilde{m}'_B)$.
        ii. If $\mathcal{K}_{N,\tilde{m}}$ is too small, recursively go to Step 2(c).
        iii. Else, if $C(N, \tilde{m}) < C_{\min}$, set $(m, C_{\min}) \leftarrow (\tilde{m}, C(N, \tilde{m}))$.
3. If $C_{\min}$ was updated in Step 2, then repeat Step 2.
4. Return $(m, C_{\min})$.

This algorithm applies small changes to the bound vector m at each step, reducing one entry while possibly increasing others. Obviously, this finds only a *locally* optimal m with respect to these changes and the initial choice of m in Step 1; different choices generally produce different results.

One way to choose an initial m is to try $(1, 1, \dots, 1)$, then $(2, 2, \dots, 2)$, etc., stopping when $\mathcal{K}_{N,m}$ is large enough. Another approach, in the context of the N search described below, is to start from the best m found for the parent N, and merely increase the first component of m until $\mathcal{K}_{N,m}$ is large enough; usually at most one increase is needed.

The algorithm involves at least $B(B-1)$ evaluations of the cost function for the final pass through Step 2. It can involve many more evaluations if there are many recursive calls or if there are many improvements to m, but usually these are small effects.

Optimizing the prime batches. We optimize N via a similar greedy algorithm, using the algorithm above as a subroutine. For a *fixed* number of batches B, we proceed as follows:

1. Choose an initial $N = (N_1, \dots, N_B)$ with $\sum_i N_i = n$, and let $(m, C_{\min})$ be the output of the algorithm above applied to N.
2. For each $i \in \{1, \dots, B\}$, do the following:

   (a) Set $\tilde{N}^i \leftarrow (N_1, \dots, N_{i-1}, N_i - 1, N_{i+1}, \dots, N_B)$.
   (b) For each $j \neq i$ in $\{1, \dots, B\}$,
      i. Set $\tilde{N}^{i,j} \leftarrow (\tilde{N}^i_1, \dots, \tilde{N}^i_{j-1}, \tilde{N}^i_j + 1, \tilde{N}^i_{j+1}, \dots, \tilde{N}^i_B)$.
      ii. Let $(\tilde{m}, \tilde{C})$ be the output of the algorithm above applied to $N^{i,j}$.
      iii. If $\tilde{C} < C_{\min}$, then update $(N, m, C_{\min}) \leftarrow (\tilde{N}^{i,j}, \tilde{m}, \tilde{C})$.

3. If $C_{\min}$ was updated in Step 2, then repeat Step 2.

4. Return N, m, and $C_{\min}$.

This algorithm also finds only a local optimum with respect to these changes, and with respect to the initial choice of N in Step 1; again, different choices may lead to different results. Our experiments took an initial choice for N such that $z \le N_1 \le \cdots \le N_B \le z + 1$ for some $z \in \mathbb{Z}$. One can also omit one or more large primes ℓ_j by taking each $N_j = 1$ and $m_j = 0$.

Within the full two-layer greedy algorithm, each N considered at the upper layer involves $B(B-1)$ calls to the lower layer, the optimization of 1-norm bounds. Recall that each call to the lower layer involves at least $B(B-1)$ evaluations of the cost function. Overall there are nearly B^4 evaluations of the cost function.

Numerical examples. Table 4.3 shows examples of outputs of the above search. For each B, the "N"/"m" column shows the final (N, m) found, and the "cost" column shows the cost function for that (N, m), to two digits after the decimal point.

We used a server with two 64-core AMD EPYC 7742 CPUs, but limited each search to 32 cores running in parallel. We parallelized only the upper layer of the search; often fewer than 32 cores were used since some calls to the lower layer were slower than others. For each B, "wall" shows the seconds of real time used for the search, and "CPU" shows the total seconds of CPU time (across all cores, user time plus system time) used for the search.

4.2.7 Constant-time software for the action

We have built a self-contained high-performance software package, `high-ctidh`, that includes implementations of all of the operations needed by CSIDH users for whichever parameter set is selected: constant-time key generation, constant-time computation of the CSIDH action, and validation of (claimed) public keys. The package uses the CTIDH key space and CTIDH algorithms to set new cycle-count records for constant-time CSIDH.

The `high-ctidh` source code is in C, with assembly language for field arithmetic. Beyond the performance benefits, using low-level languages is helpful for ensuring constant-time behavior, as explained below. Measuring

B	wall	cost	N
	CPU		m
1	1.27	3462230.00	74
	1.25		153
2	1.55	1483388.79	36 38
	1.80		64 96
3	2.59	990766.14	23 27 24
	4.78		40 62 62
4	5.14	755266.87	14 19 20 21
	22.17		29 45 46 46
5	4.65	649002.35	13 15 15 17 14
	22.15		23 37 38 38 35
6	13.29	583256.02	10 11 12 12 15 14
	150.29		19 31 31 32 32 30
7	24.98	537496.27	7 10 10 12 12 14 9
	334.31		17 27 28 28 28 28 22
8	65.90	504984.23	5 9 9 10 10 11 11 9
	1141.82		15 24 25 25 25 26 26 16
9	138.65	485052.29	5 7 8 8 8 7 10 12 9
	2763.28		14 22 23 23 23 23 24 24 13
10	393.63	471184.70	5 7 8 8 8 7 9 10 11 1
	8209.63		13 20 22 22 22 22 22 22 22 1
11	966.91	451105.76	3 5 6 7 7 8 7 9 10 11 1
	21740.60		11 18 19 20 20 21 20 21 21 21 1
12	1484.31	448573.04	3 4 6 6 6 6 7 7 8 10 10 1
	36763.23		10 16 18 18 19 19 19 19 19 19 19 2
13	2301.94	445054.10	3 4 4 6 6 6 7 7 8 7 7 8 1
	55252.51		10 16 17 18 18 18 18 18 18 18 17 14 1
14	6509	437985.55	2 3 4 4 5 5 6 7 7 8 8 6 8 1
	161371.00		10 14 16 17 17 17 18 18 18 18 18 13 13 1
15	8341	440201.56	3 4 3 4 4 5 5 5 6 6 6 7 7 8 1
	211336.80		9 14 15 15 16 16 16 16 16 16 16 16 15 13 1
16	18060	442718.29	2 3 4 4 5 5 5 5 6 6 7 8 4 1 8 1
	491547.34		9 13 15 16 16 17 17 17 17 16 17 17 7 1 16 1
17	29733	450343.88	2 3 4 4 5 5 5 5 5 5 5 5 7 5 3 5 1
	808639.64		8 12 14 15 15 15 16 15 16 16 14 13 16 11 6 10 1
18	73925	443412.54	2 2 3 3 4 4 5 5 5 5 5 6 6 6 3 1 8 1
	2012125.98		8 11 13 14 14 15 15 15 15 15 15 15 14 14 7 2 15 1
19	103825	447506.32	2 2 3 3 3 4 4 5 5 5 6 4 6 1 8 4 7 1 1
	2961123.56		8 12 14 15 15 15 15 16 16 16 16 10 16 2 16 7 14 1 1
20	167114	455328.80	2 2 3 3 4 4 5 5 5 5 5 4 7 7 1 3 6 1 1 1
	4794006.52		9 12 14 14 16 16 16 16 16 16 16 11 16 16 2 5 12 1 1 1
21	278646	460901.00	2 2 3 3 4 4 5 5 5 5 5 7 2 1 1 3 1 7 7 1 1
	7981372.99		9 13 15 16 16 16 16 17 17 17 16 17 5 2 2 6 1 17 11 1 1

Table 4.3: Results of searches, for various choices of B, for CTIDH parameters with at least 2^{256} keys for the CSIDH-512 prime. See text for description.

the performance of a full C implementation also resolves the concerns raised by using multiplications as a predictor of performance, such as concerns that some subroutines could be difficult to handle in constant time and that improved multiplication counts could be outweighed by overhead.

The software is freely available at `https://ctidh.isogeny.org/`. This section describes the software. Section 4.2.8 reports the software speeds and compares to previous speeds.

4.2.7.1 Processor selection and field arithmetic

The original CSIDH paper reported clock cycles for variable-time CSIDH-512 software on an Intel Skylake CPU core. Skylake is also the most common CPU choice in followup papers on CSIDH software speed. We similarly focus on Skylake to maximize comparability.

The original `csidh-20180826` software from [49] included a small assembly-language library for Intel chips (Broadwell and newer) to perform arithmetic modulo the CSIDH-512 prime. The same library has been copied, with minor tweaks and generalizations to other primes, into various subsequent software packages, including `high-ctidh`. Code above the field-arithmetic level, decomposing isogenies into multiplications etc., are written in C, so porting the software to another CPU is mainly a matter of writing an efficient Montgomery multiplier for that CPU. Beware that each CPU will have different cycle counts, and possibly a different ranking of algorithmic choices.

The `velusqrt-asm` software from [19] includes an adaptation of the same library to CSIDH-1024. The `sqale-csidh-velusqrt` software from [54] includes adaptations to larger sizes, all automatically generated by a code generator that takes p as input. The `high-ctidh` package includes a similar code generator, with some small improvements in the details: for example, we use less arithmetic for conditional subtraction, and we avoid `cmov` instructions with memory operands out of concern that they could have data-dependent timings.

4.2.7.2 Computing one isogeny

The middle layer of `high-ctidh` computes an ℓ-isogeny for one prime ℓ; it also includes auxiliary functions such as multiplying by the scalar ℓ. We built this layer as follows.

We started with the `xISOG` function in `velusqrt-asm`. As in `csidh-20180826`, this function takes a curve and a point P of order ℓ, and returns the corresponding ℓ-isogenous curve. It also takes a point T, and returns the image of that point under the isogeny.

We extended the function interface to take lower and upper bounds on ℓ—the smallest and largest prime in the batch containing ℓ—and we modified the software to take time depending only on these bounds, not on the secret

ℓ. The Matryoshka-doll structure of the computation (see Section 4.2.5.5) meant that very little code had to change. Each loop to ℓ is replaced by a loop to the upper bound, with constant-time conditional selection of the results relevant to ℓ; and ℓ is replaced by the lower bound as input to the $\surd$élu parameter selection. An upper bound was used the same way in [23]; the use of the lower bound for a Matryoshka-doll $\surd$élu is new here.

We reused the automatic $\surd$élu parameter-tuning mechanisms from `velusqrt-asm`. These mechanisms offer the option of tuning for multiplication counts or tuning for cycles. Since most CSIDH-related papers report multiplication counts while fewer report cycles, we chose to tune for multiplication counts for comparability, but this makes only a small difference: cycle counts and multiplication counts are highly correlated.

We made more changes to incorporate known optimizations, including an observation from [23] regarding the applicability of multiexponentiation, and an observation from [3] regarding reciprocal polynomials. Computing a 587-isogeny and pushing a point through takes 2108 multiplications in this software (counting squarings as multiplications); for comparison, [3] took 3.4% more, and `velusqrt-asm` took 8.9% more.

More importantly for the high-level algorithms, we extended the interface to allow an array of points T to be pushed through the isogeny—e.g., two or zero points rather than one. We also incorporated shorter differential addition chains, as in [51], for scalar multiplications, and standard addition chains for the constant-time exponentiation inside Legendre-symbol computation.

There would be marginal speedups from tuning the $\surd$élu parameters separately for each number of points. Taking parameters $(6,3)$ for 0 points instead of $(0,0)$ saves 2 out of 328 multiplications for $\ell = 79$; 2 out of 344 multiplications for $\ell = 83$; and 8 out of 368 multiplications for $\ell = 89$. Parameter adjustments also save 3 multiplications for 0 points for each $\ell \in \{557, 587, 613\}$. However, we did not find such speedups for most primes, and we did not find such speedups for the much more common case of 2 points.

4.2.7.3 Computing the action

The top layer of `high-ctidh` is new, and includes the core CTIDH algorithms described earlier in this paper. The key space is $\mathcal{K}_{N,m}$, allowing any vector with 1-norm at most m_1 for the first N_1 primes, 1-norm at most m_2 for the first N_2 primes, etc. Constant-time generation of a length-N_i vector of 1-norm at most m_i works as follows:

- Generate $N_i + m_i$ uniform random b-bit integers.
- Set the bottom bit of each of the first N_i integers, and clear the bottom bit of each of the last m_i integers.

- Sort the integers. (We reused existing constant-time sorting software from [16].)
- If any adjacent integers are the same outside the bottom bit, start over. (Otherwise the integers were distinct outside the bottom bit, so sorting them applies a uniform random permutation.)
- Extract the bottom bit at each position. (This is a uniform random bit string of length $N_i + m_i$ with exactly N_i bits set.)
- Consider the entries as integers. Add the first entry to the second, then add the resulting second entry to the third, etc. (Now there are maybe some 0s, then at least one 1, then at least one 2, and so on through at least one N_i.)
- Count, in constant time, the number e_0 of 0, the number e_1 of 1, and so on through the number e_{N_i-1} of $N_i - 1$. (These tallies add up to at most $N_i + m_i - 1$, since the number of N_i was not included. Each of $e_1, \dots, e_{N_i-1}$ is positive, and e_0 is nonnegative.)
- Subtract 1 from each of $e_1, \dots, e_{N_i-1}$. (Now $e_0, \dots, e_{N_i-1}$ is a uniform random string of N_i nonnegative integers with sum at most m_i.)
- Generate a uniform random N_i-bit string $s_0, \dots, s_{N_i-1}$.
- Compute, in constant time, whether any j has $s_j = 1$ and $e_j = 0$. If so, start over.
- Replace each e_j with $-e_j$ if $s_j = 1$.

As required by the constant-time property, the two rejection steps in this algorithm are independent of the secrets produced as output. The first rejection step is very unlikely to occur when b is chosen so that 2^b is on a larger scale than $(N_i + m_i)^2$. The second rejection step occurs more frequently. Sign variations for vectors of Hamming weight k contribute 2^k by Lemma 1 and thus the rejection correctly happens more frequently for smaller k.

In the case $N_i > m_i$, `high-ctidh` saves time by skipping (in constant time) the $s_j = 1$ rejection test for the first $N_i - m_i$ values of j having $e_j = 0$. There are always at least $N_i - m_i$ such values of j. This increases each acceptance chance by a factor $2^{N_i - m_i}$, preserving uniformity of the final output.

Once a private key is generated, the action is computed by a series of restricted square-free ABs. As in Section 4.2.4, the first AB handles one prime from each batch, the next AB handles one prime from each batch that might have something left to do, etc.

Within each AB, Elligator is used twice to generate two independent points; see Appendix. Specifically, Elligator is used to generate a point on the first curve E_A: a point in $\tilde{E}_A(\mathbb{F}_p)$ if the first isogeny has negative sign, otherwise in $E_A(\mathbb{F}_p)$. This point is pushed through the first isogeny. Elligator is then used again to generate an independent point on the second curve $E_{A'}$: a point in $\tilde{E}_{A'}(\mathbb{F}_p)$ if the first isogeny had positive sign, otherwise in $E_{A'}(\mathbb{F}_p)$. Both choices are secret. These two points (T_0, T_1) are then pushed through subsequent isogenies as in Algorithm 10, except that no points are pushed through the last isogeny and only one point is pushed through the isogeny before that. The AB thus pushes $1, 2, 2, 2, \ldots, 2, 2, 2, 1, 0$ points through isogenies. The software permutes the $b \leq B$ batches in the AB to use primes $\ell_{b-1}, \ell_{b-3}, \ell_{b-4}, \ldots, \ell_1, \ell_{b-2}, \ell_b$ in that order.

Each AB selects one prime from each batch in the block and tries to compute an isogeny of total degree D, the product of the selected primes; $D = r'$ in Algorithm 10. Each point is multiplied by 4 and then by all primes outside D immediately after the point is generated by Elligator, so that the order of the point divides D. There are two types of primes outside D (compare Steps 2 and 3 of Algorithm 10):

- The batches in the AB are public. Primes outside these batches are publicly outside D.
- Primes that are inside the batches in the AB, but that are not the secretly selected prime per batch, are secretly outside D.

For scalar multiplication by a product of secret primes, [23] uses a Montgomery ladder, with the number of ladder steps determined by the maximum possible product. For public primes, [51] does better using a precomputed differential addition chain for each prime. Our `high-ctidh` software also uses these chains for secret primes, taking care to handle the incompleteness of differential-addition formulas and to do everything in constant time. The primes in a batch usually vary slightly in chain length, so the software always runs to the maximum length.

Each ℓ-isogeny then clears ℓ from the order of the point that was used to compute the isogeny. As in line 14 of Algorithm 10, the software multiplies the point by ℓ anyway (again using a constant-time differential addition chain), just in case this was a dummy isogeny, i.e., there was secretly nothing left to do in the batch. This extra scalar multiplication could be merged with the isogeny computation, but the $\sqrt{}$élu structure seems to make this somewhat more complicated than in [136], and the extra scalar multiplication accounts for only about 3% of the CSIDH-512 computation. The other point is also multiplied by ℓ.

Recall that an AB successfully handling a batch is a public event, visible in timing: it means that a (real or dummy) ℓ-isogeny is computed now for some ℓ in the batch, publicly decreasing the maximum 1-norm of the batch. This event occurs with probability $1 - 1/\ell_{i,1}$, where $\ell_{i,1}$ is the smallest prime in the batch containing $\ell = \ell_{i,j}$. As in Section 4.2.5.4, the software creates this event exactly when there is a conjunction of a natural success and an artificial success. A natural success, probability $1 - 1/\ell$, means that cofactor multiplication produces a point of order ℓ rather than order 1. An artificial success, probability $\gamma = (1 - 1/\ell_{i,1})/(1 - 1/\ell)$, is determined by a γ-biased coin toss.

One obvious way to generate a γ-biased coin is to (1) generate a uniform random integer modulo $\ell_{i,1}(\ell - 1)$ and (2) compute whether the integer is smaller than $\ell(\ell_{i,1} - 1)$. The second step is easy to do in constant time. For the first step, the software generates a uniform random 256-bit integer and, in constant time, reduces that modulo $\ell_{i,1}(\ell - 1)$; the resulting distribution is indistinguishable from uniform. One could instead use rejection sampling to compute a uniform random integer modulo $\ell_{i,1}M$, where M is the least common multiple of $\ell - 1$ across primes ℓ in the batch, and then reduce the integer modulo $\ell_{i,1}(\ell - 1)$, to obtain an exactly uniform distribution; the reason to use M here rather than just one $\ell - 1$ is to avoid having the secret ℓ influence the rejection probability.

4.2.7.4 Automated constant-time verification

We designed and analyzed every step of the CTIDH algorithm to be constant time, leaking nothing about the input through timing; this is the basis for our claim that the algorithm is in fact constant time. We also designed and reviewed every new line of code in `high-ctidh` to be constant time, and reviewed every line of reused code for the same property; this is the basis for our claim that the software is in fact constant time. These analyses are complete—but, as in most papers on constant-time algorithms, are entirely done by hand, raising the question of what protections there are against human error.

For extra assurance, we designated an internal auditor to use automated tools to verify the constant-time claims. This subsection is an audit report to support external auditing. This report describes what the tools verified, describes various limitations of this verification, and describes various steps that the auditor took to compensate for those limitations.

From a risk-management perspective, a timing leak in `high-ctidh` would have to be at the intersection of (1) human error in this paper's manual analysis and (2) limitations of the automated verification. One might hope for automated verification without any limitations, eliminating the need for manual analysis (assuming correctness of the verification tools), but one risk identified below is beyond the current state of the art in automated verification. The automated verification is nevertheless useful in reducing risks overall.

An automated test using `valgrind`. There is a standard tool, `valgrind` [151], that runs a specified binary, watching each instruction for memory errors—in particular, branches and array indices derived from undefined data. If secret data in cryptographic software is marked as undefined then simply running `valgrind` will automatically check whether there is any data flow from secrets to branches and array indices; see, e.g., [121]. See also [107] for a survey of related tools.

Because `valgrind` works at the binary level, this analysis includes any optimizations that might have been introduced by the compiler. A compiler change could generate a different binary with timing leaks, but `valgrind` is fast enough to be systematically run on all compiled cryptographic software before the software is deployed.

The auditor wrote a simple `checkct` program using `high-ctidh` to perform a full CSIDH key exchange; this program is included in the `high-ctidh` package. For example, running `valgrind ./checkct512default` takes under 30 seconds on a 3GHz Skylake core, where `checkct512default` performs a full CSIDH-512 key exchange. The underlying `randombytes` function marks all of its output as undefined, so `valgrind` is checking for any possible data flow from randomness to branches or to array indices. For each size, `valgrind` completes successfully, indicating that there is no such data flow.

Limitations of the automated test, and steps to address the limitations. The following paragraphs ask, from the auditor's perspective, what could have been missed *by this automated test*—for example, the auditor asks what would happen if private keys were actually generated by OpenSSL's `RAND_bytes` rather than `randombytes`. Everything is covered by the paper's manual analysis—for example, we had already checked that all code was generating all randomness via `randombytes`—but the question addressed here is the level of *extra* assurance provided by the automated analysis.

If the code generates randomness such as private keys via `RAND_bytes` rather than `randombytes`, then private keys will not be marked as undefined, so `valgrind` will not track data flow from private keys to branches or to array indices. To address this, the auditor skimmed `high-ctidh` to check the (very limited) set of C library functions being used, and double-checked the list of functions output by `nm checkct512default`.

If private keys are actually deterministic then they will not be marked as undefined. To address this, the auditor added a step to `checkct` to mark Alice's private key as undefined before Alice handles Bob's public key. The auditor also checked examples of private keys and saw them varying.

The `valgrind` analysis is dynamic, tracing through one run of code. Perhaps CSIDH-1024 triggers code paths that are not used by CSIDH-512 and that leak secret data. To address this, the auditor tried all sizes of interest.

Different runs could still follow different code paths because of the declassification described below. To address this, the auditor tried many runs of each size, but there is still a risk that all of the runs missed some code path. In theory it should be possible to combine `valgrind` with static code-coverage analysis for binaries that follow standard calling conventions, but as far as we know no tools are available for this. There are tools for static constant-time analysis of C code and binaries [106], and those tools could be applied to a modified version of `high-ctidh` that replaces the assembly-language portions with reference C code.

The `valgrind` analysis checks that all array indices and all branch conditions are defined, but does not check that division inputs are defined. Division instructions take variable time in most CPUs, and should *not* be modeled as taking constant time. To address this, the auditor skimmed the assembly-language code for any use of division instructions, and skimmed the C code for any operations likely to be compiled into division instructions. Patching `valgrind` to limit the set of acceptable instructions would reduce risks here.

Finally, the `high-ctidh` package has six `crypto_declassify` lines explicitly marking certain pieces of data as *defined*, meaning that `valgrind` allows branches and array indices derived from that data. Perhaps this declassification leaks secrets. This is the most important risk, the risk that would require advances in automated verification to address.

Five of the six lines are in rejection-sampling loops: one in generating uniform random integers modulo p (rejecting numbers $\geq p$), two in Elligator (rejecting random numbers $0, 1, -1$), and two in the subroutine described above to generate vectors of bounded 1-norm. The sixth, and the most worrisome, declassifies the success of a batch in an AB. Analyzing the safety of this declassification requires analyzing everything that influences the success probability, including

- the logic concluding that the natural failure probability of generating a curve point of order $\ell_{i,j}$ is exactly $1/\ell_{i,j}$,
- the coin toss artificially increasing the failure probability to exactly $1/\ell_{i,1}$, and
- the use of Elligator as a substitute for uniform random points, assuming (as previously conjectured; see Appendix) indistinguishability of the point orders.

We again emphasize that all of this analysis is included in this paper. The challenge for the future is to *automate* the analysis.

4.2.8 Software speeds

This section reports various measurements of the `high-ctidh` software from Section 4.2.7, and compares the measurements to previous speeds for constant-time CSIDH.

4.2.8.1 Selecting a CSIDH size and collecting performance data

For comparability to previous speed reports, we focus here on CSIDH-512 with a key space of 2^{256} vectors. After some searching we took the (N, m) shown for $B = 14$ in Figure 4.3. This (N, m) has approximately $2^{256.009}$ keys. Our cost calculator claimed that this (N, m) would use approximately 437986 multiplications on average.

We chose parameters a and c, and performed a different action computations for each of c different private keys on a 3GHz Intel Xeon E3-1220 v5 (Skylake) CPU with Turbo Boost disabled. This CPU does not support hyperthreading, and to limit noise we used only one core. For each of the ac computations, we recorded a cycle count, a total multiplication count including squarings, a separate count of squarings, and a total addition count including subtractions. We also tracked, for each key and each batch of primes, the success probability of that batch in ABs for the computations for that key.

Choosing $c = 65$, as in [19], and $a = 16383$ meant that experiments completed quickly, half a day on one core. We did not detect any deviations from the null hypothesis that the software performance is independent of the private key. For example, as discussed below, the per-key success probability of the batch having smallest prime $\ell_{i,1}$ was not statistically distinguishable from $1 - 1/\ell_{i,1}$.

One can easily justify spending further computer time on experiments. Larger a or c would make the total statistics more robust. Larger a would make the per-key statistics more robust. Larger c would be useful if there were a set of, say, 1 in every 1000 keys that somehow leaked information through timing. On the other hand, predictable CTIDH implementation errors such as taking a coin with probability $1 - \gamma$ rather than γ would have been caught by our experiments.

4.2.8.2 Performance results for the selected CSIDH size

We use the standard notation $\mathbf{M}$ for multiplications not including squarings, $\mathbf{S}$ for squarings, and $\mathbf{a}$ for additions including subtractions. One common metric in the literature is $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$, counting the total number of multiplications while ignoring the costs of addition and ignoring possible squaring speedups. Another common metric is $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$.

Across all 1064895 experiments, the average cycle count was 125.53 million, standard deviation 3.01 million. The average $\mathbf{M}$ was 321207, standard deviation 6621. The average $\mathbf{S}$ was 116798, standard deviation 4336. The average $\mathbf{a}$ was 482311, standard deviation 9322. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 438006. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 438762.

For the first key in particular, the averages were 125.55 million, 321270, 116837, and 482399 respectively. The gaps between these per-key averages and the overall averages are +0.90%, +0.94%, +0.89%, +0.94%, respectively, of a standard deviation, which is unsurprising for 16383 experiments per key. The gaps for the next four keys are −0.61%, −0.58%, −0.56%, −0.59%, −0.06%, −0.02%, −0.02%, −0.02%, −1.44%, −1.39%, −1.37%, −1.39%, +0.09%, +0.20%, +0.19%, +0.20%, respectively, of a standard deviation. The per-batch success probability for the first key, divided by the expected $1 - 1/\ell_{i,1}$, was 0.999670 for the first batch, 1.000785 for the second batch, 0.999688 for the third batch, etc.; for the second key, 1.001005, 0.998374, 1.000442, etc.; for the third key, 0.999304, 1.000131, 0.999820, etc.; for the fourth key, 1.000476, 0.998714, 0.999224, etc.; for the fifth key, 1.001030, 1.001379, 1.000829, etc. The first-batch gaps from the predicted average (namely 1) for the first 10 keys are −0.05%, +0.14%, −0.10%, +0.07%, +0.15%, −0.35%, +0.13%, +0.01%, +0.12%, −0.21% of the predicted standard deviation (namely $\sqrt{1/2}$); note that each of the 16383 experiments involves around 20 first-batch tries.

To understand the performance results in more detail, we plotted the distribution of all *ac* multiplication counts as the red curve in Figure 4.2. We also computed, for each key, the distribution of the 16383 multiplication counts for that key; there are five blue curves in Figure 4.2, showing the minimum, first quartile, median, third quartile, and maximum of these 65 distributions. The green curves, with a larger spread, are like the blue curves but are limited to the last 255 multiplication counts for each key.

Each curve has a stair-step shape. Another step upwards reflects another AB in the computation, with (typically) two Elligator calls and two large scalar multiplications. Any number of ABs *can* appear—for example, $\ell = 3$ can fail again and again—but with exponentially low probability. One can extrapolate the budget needed for an application that needs to run for a fixed time (e.g., [23]) with a failure probability of, e.g., 2^{-100}; in this scenario the time would be lower if the smallest primes were left out of all batches.

It is unsurprising to see that the green curves have a spread of step positioning on the scale of 10%, given that these curves consider only 255 experiments for each of the 65 keys. Similar comments apply to the blue curves, with more experiments and a narrower spread. As a visual illustration that the spread is what one would expect, Figure 4.3 replaces the green and blue curves from Figure 4.2 with random simulations based on the red curve.

To gain more confidence that the distributions match, one could run more experiments for each key and watch for a further narrowing from the blue curves towards the red curve. Note that graphing the complete distributions provides much more information than computing a single number such as a t-statistic. Similar comments apply to cost metrics beyond multiplications: for example, Figure 4.4 shows cycle counts, and one could similarly plot other statistics such as the time of the first failed batch.

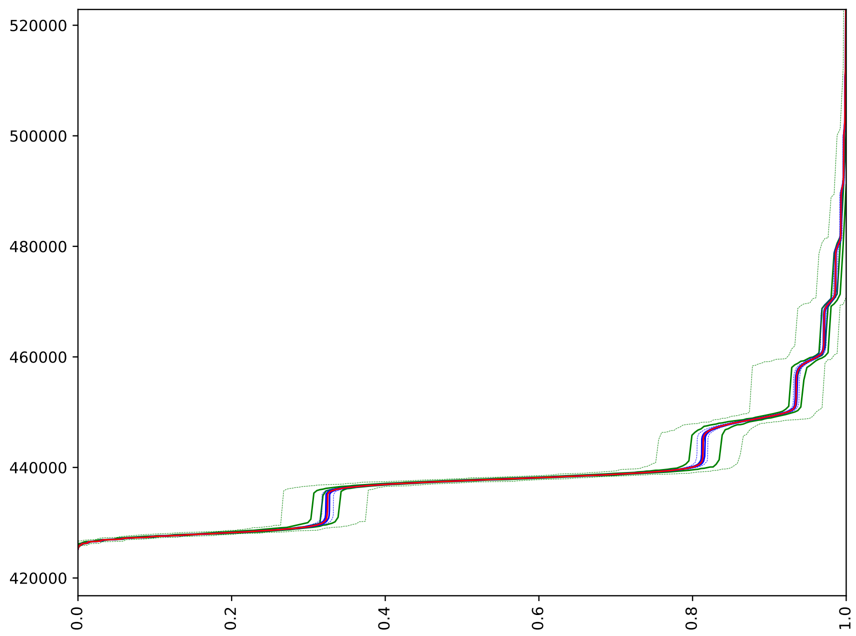


Figure 4.2: Distributions of costs in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric, i.e., total multiplication counts. Five green curves: minimum, first quartile, median, third quartile, and maximum of 65 per-key distributions of multiplication counts in the last 255 experiments. Five blue curves: minimum, first quartile, median, third quartile, and maximum of 65 per-key distributions of multiplication counts in all 16383 experiments. Red curve: distribution of multiplication counts in all experiments across all keys.

We also measured the cost of validation of a public key: median 14680 multiplications (after some easy speedups), around 4.09 million cycles. Note that validation takes variable time: an invalid key fails much more quickly. Finally, we measured the cost of generating a private key: typically under 1 million cycles, a negligible cost compared to generating the corresponding public key.

4.2.8.3 Other CSIDH sizes

We also evaluated two further sizes, running $65 \cdot 16383$ experiments per size, to illustrate performance variations in two different dimensions of the CSIDH parameter space.

First, to understand the effect of having a larger list of ℓ, we switched from the CSIDH-512 prime to the CSIDH-1024 prime, while keeping the

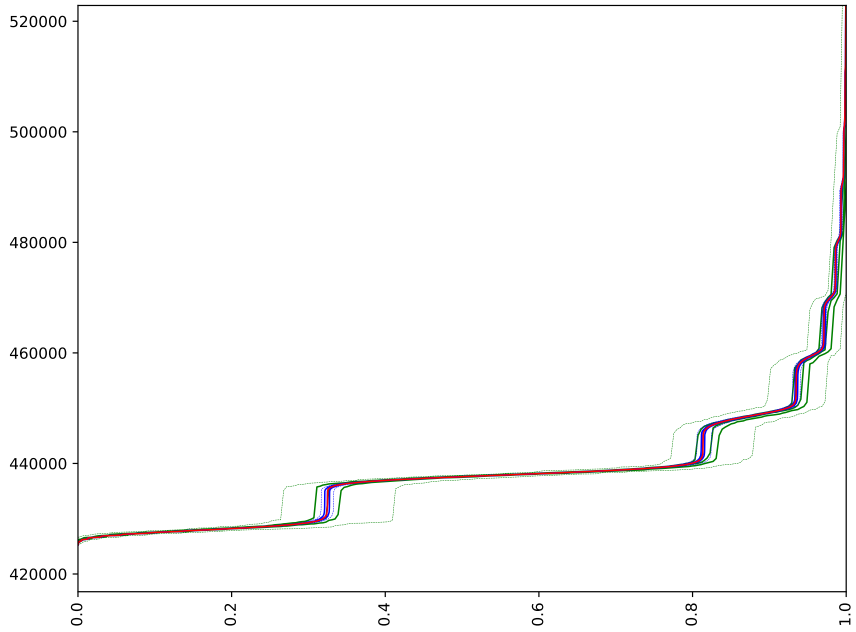


Figure 4.3: Simulation of Figure 4.2. The red curve is copied from Figure 4.2. The green and blue curves replace the underlying per-key data with random samples from the red curve.

same size of key space. After some searching we took CTIDH batch sizes

$$2, 3, 5, 4, 6, 6, 6, 6, 6, 7, 7, 7, 7, 6, 7, 7, 5, 6, 5, 10, 3, 10, 5, 1, \text{ with bounds}$$

$$2, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 3, 6, 2, 6, 2, 0.$$

There are approximately $2^{256.066}$ keys.

Across all 1064895 experiments, the average cycle count was 469.52 million, standard deviation 15.29 million. The average $\mathbf{M}$ was 287739, standard deviation 7420. The average $\mathbf{S}$ was 87944, standard deviation 4961. The average $\mathbf{a}$ was 486764, standard deviation 10525. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 375683. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 382432.

As in [3, Tables 1 and 2], the CSIDH-1024 prime uses fewer multiplications than the CSIDH-512 prime (although, unsurprisingly, the larger prime makes each multiplication slower). One might think that a larger list of ℓ needs more multiplications, since one needs to clear more cofactors; but the larger list also means that one can use smaller exponents for the same size key space, and in our experiments this turns out to have a larger effect.

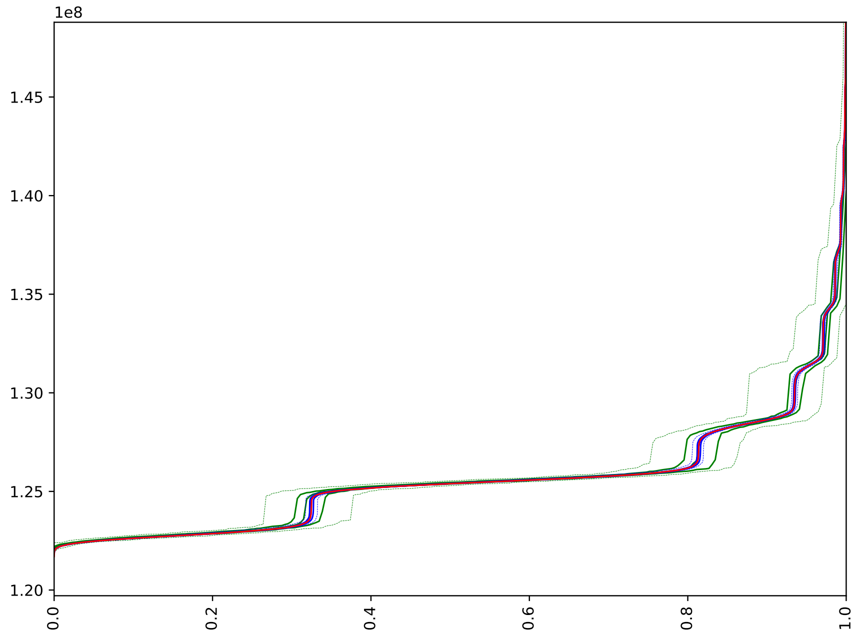


Figure 4.4: Same as Figure 4.2, but replacing multiplication counts with cycle counts scaled by 10^8.

Second, to understand the effect of changing the size of the key space, we took CSIDH-512 with only 2^{220} keys, as in the software from [54]. Specifically, we took CTIDH batch sizes $2, 3, 4, 4, 5, 5, 5, 5, 5, 7, 7, 8, 7, 6, 1$, with bounds $6, 9, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 8, 6, 1$. There are approximately $2^{220.004}$ keys. This option is labeled “511” in `high-ctidh`.

Across all 1064895 experiments, the average cycle count was 89.11 million, standard deviation 2.37 million. The average $\mathbf{M}$ was 228780, standard deviation 5186. The average $\mathbf{S}$ was 82165, standard deviation 3428. The average $\mathbf{a}$ was 346798, standard deviation 7344. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 1, 0)$ metric was 310945. The average cost in the $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$ metric was 311852.

4.2.8.4 Comparisons

There have been several previous speed reports [136, 154, 51, 104, 56, 3, 54] for constant-time CSIDH. CSIDH-512 with a key space of 2^{256} vectors is almost always included, and for this size the lowest multiplication count we have found in the literature is 789000: this is from [3, Table 1, “hvelu”, “OAYT-style”], which reports $624000\mathbf{M} + 165000\mathbf{S} + 893000\mathbf{a}$. All Skylake cycle counts we have found are above 200 million. The high-ctidh speeds are much faster, and have the added feature of constant-time verification using valgrind.

Sometimes the latest software speeds are not fully reflected in the relevant papers, so we downloaded the latest versions of csidh_withstrategies (dated July 2020) from [56] and sqale-csidh-velusqrt (dated December 2020) from [54], and ran their benchmarking tools—with one modification to change the number of experiments ("its") from 1024 to 65536—on the same Skylake machine that we had used for measuring high-ctidh. Some care is required in comparisons, for at least three reasons: first, some tools report the time for an action *plus* key validation; second, different benchmarking frameworks could be measuring different things (e.g., our impression is that the costs of Elligator were omitted from the multiplication counts reported in [3]); third, the 512-bit parameters in sqale-csidh-velusqrt use a key space of size only 2^{220}, as noted above. Note that for CSIDH-1024 there is even more variation in the literature in the size of key space; e.g., the original CSIDH-1024 software from [49] used $5^{130} > 2^{300}$ keys.

The csidh_withstrategies tools, using BITLENGTH_OF_P=512 TYPE=WITHDUMMY_2 APPROACH=STRATEGY, reported averages of 218.42 million clock cycles (standard deviation 3.39 million), 691231$\mathbf{a}$ (standard deviation 12554), 189377$\mathbf{S}$ (standard deviation 4450), and 665876$\mathbf{M}$ (standard deviation 7888); in other words, 855253 multiplications, or 851939 counting $(\mathbf{M}, \mathbf{S}, \mathbf{a}) = (1, 0.8, 0.05)$.

The sqale-csidh-velusqrt tools, using BITS=512 STYLE=wd2, reported averages of 190.921 million cycles (standard deviation 4.32 million), 626000$\mathbf{a}$ (standard deviation 13000), 128000$\mathbf{S}$ (standard deviation 5000), and 447000$\mathbf{M}$ (standard deviation 9000); i.e., 575000 multiplications. For comparison, high-ctidh takes 89.11 million cycles (310945 multiplications) as noted above, plus 4.09 million cycles for validation.

pub	priv	DH	Mcyc	**M**	**S**	**a**	$1, 1, 0$	$1, 0.8, 0.05$	
512	220	1	89.11	228780	82165	346798	310945	311852	new
512	220	1	190.92	447000	128000	626000	575000	580700	[54]
512	220	2	93.23	238538	87154	361964	325692	326359	new
512	256	1	125.53	321207	116798	482311	438006	438762	new
512	256	1	—	624000	165000	893000	789000	800650	[3]
512	256	2	129.64	330966	121787	497476	452752	453269	new
512	256	2	218.42	665876	189377	691231	855253	851939	[56]
512	256	2	238.51	632444	209310	704576	841754	835121	[104]
512	256	2	239.00	657000	210000	691000	867000	859550	[51]
512	256	2	—	732966	243838	680801	976804	962076	[154]
512	256	2	395.00	1054000	410000	1053000	1464000	1434650	[136]
1024	256	1	469.52	287739	87944	486764	375683	382432	new
1024	256	1	—	552000	133000	924000	685000	704600	[3]
1024	256	2	511.19	310154	99371	521400	409525	415721	new

Table 4.4: Comparison of speed reports for constant-time CSIDH actions. The CSIDH size is specified by "pub" (512 for the CSIDH-512 prime, 1024 for the CSIDH-1024 prime) and "priv" (k where private keys are chosen from a space of approximately 2^k vectors). "DH" is the Diffie–Hellman stage: "1" for computing a public key (computing the CSIDH action), "2" for computing a shared secret (validating a public key and then computing the CSIDH action). "Mcyc" is millions of Skylake cycles (not shown for Python software); "**M**" is the number of multiplications not including squarings; "**S**" is the number of squarings; "**a**" is the number of additions including subtractions; "$1, 1, 0$" and "$1, 0.8, 0.05$" are combinations of **M**, **S**, and **a**. See text for measurement details and standard deviations.

Finally, Table 4.4 summarizes the measurements listed above for `high-ctidh`, for the software from [56], and for the software from [54]; the measurements stated in [154, 51, 104, 3] for the software in those papers; and the measurements stated in [51] for the software in [136]. For [104] the reported processor is an Intel Core i7-7500k, which is Kaby Lake rather than Skylake, but Kaby Lake cycle counts generally match Skylake cycle counts. The table omits cycle counts for [3], which used Python, and [154], which used C but had measurements affected by an unknown amount of Turbo Boost.

4.2.9 Appendix

The main body of this chapter focuses on protecting against timing attacks. This appendix considers the extra challenge of protecting against faults.

Fault-injection attacks on constant-time CSIDH implementations are discussed in [51, 40]. Dummy operations are dangerous in this context: a "safe-error attack" faults an operation and, if the output is unchanged, concludes that the operation was a dummy operation. The literature thus aims for "dummy-free" algorithms as a step towards protecting against faults. A dummy-free constant-time group-action algorithm, based on the 2-point approach of [154], was proposed in [51]. This algorithm uses the modified key space

$$\tilde{\mathcal{K}}_m := \prod_{i=1}^{n}\{-m_i, -m_i+2, \ldots, m_i-2, m_i\} \qquad \text{with} \qquad \#\tilde{\mathcal{K}}_m = \prod_{i=1}^{n}(m_i+1)\,.$$

The action evaluation computes e_i isogenies of degree ℓ_i as usual, followed by $(m_i - |e_i|)/2$ isogenies in both the positive *and* negative directions. These isogenies effectively cancel each other, and we obtain the same resulting curve as when computing $|e_i|$ isogenies and $m_i - |e_i|$ dummy isogenies. This requires a total of m_i isogenies per degree, as in [154], but m_i has to be chosen twice as large for the same size of key space, so overall the dummy elimination costs a factor 2.

Algorithm 11 presents the dummy-free group action from [51] in terms of dummy-free ABs—which are just the usual squarefree ABs from Section 4.2.5, but with $R = \{-1, 1\}$ (so that no dummy isogenies are computed). The similarity to Algorithm 7 is clear.

CTIDH adapts easily to a dummy-free variant. Algorithm 12, a generalization of Algorithm 11 for $\tilde{\mathcal{K}}_m$, uses restricted square-free ABs with $R = \{-1, 1\}$ to handle keys in

$$\tilde{\mathcal{K}}_{N,m} := \left\{(e_1, \ldots, e_n) \in \mathcal{K}_{N,m} \mid \textstyle\sum_{j=1}^{N_i} |e_{i,j}| \equiv m_i \pmod 2 \text{ for all } i\right\},$$

a batching-oriented generalization of $\tilde{\mathcal{K}}_m$. We have $\#\tilde{\mathcal{K}}_{N,m} = \prod_{i=1}^{B} \tilde{\Phi}(N_i, m_i)$, where $\tilde{\Phi}$ sums $\Phi(N_i, j) - \Phi(N_i, j-1)$ for $j = m_i$, $j = m_i - 2$, etc., analogously to Lemma 1.

Algorithm 11: The dummy-free constant-time group action evaluation from [51].

```
Parameters: $m = (m_1, \dots, m_n)$
Input: $A \in \mathcal{M}$, $e = (e_1, \dots, e_n) \in \tilde{\mathcal{K}}_m$
Output: $A'$ with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$
1  $(\mu_1, \dots, \mu_n) \leftarrow (m_1, \dots, m_n)$ ;
2  while $(\mu_1, \dots, \mu_n) \neq (0, \dots, 0)$ do
3      Let $I = (I_1, \dots, I_k)$ s.t. $I_1 < \dots < I_k$ and
         $\{I_1, \dots, I_k\} = \{1 \le i \le n \mid \mu_i > 0\}$ ;
4      Choose $e^+ \in \mathbb{Z}^n_{\ge 0}$ and $e^- \in \mathbb{Z}^n_{\le 0}$ such that $e_i^+ + e_i^- = e_i$ and
         $|e_i^+| + |e_i^-| = m_i$ for $1 \le i \le n$ ;
5      for $1 \le i \le k$ do
6          $\epsilon_i \leftarrow \begin{cases} 1 & \text{if } e^+_{I_i} \neq 0 \\ -1 & \text{if } e^+_{I_1} = 0 \end{cases}$
7      $(A, f) \leftarrow \alpha_{R,I}(A, (\epsilon_1, \dots, \epsilon_k))$ ;                // Square-free AB
8      for $1 \le i \le k$ do
9          $\mu_{I_i} \leftarrow \mu_{I_i} - f_i$ ;
10         if $\epsilon_i = 1$ then
11             $e^+_{I_i} \leftarrow e^+_{I_i} - \epsilon_i \cdot f_i$
12         else
13             $e^-_{I_i} \leftarrow e^-_{I_i} - \epsilon_i \cdot f_i$
14 return $A$
```

Algorithm 12: A constant-time group action for keys in $\tilde{\mathcal{K}}_{N,m}$ based on restricted squarefree ABs with $R = \{-1, 1\}$.

```
   Parameters: $N$, $m$, $B$
   Input: $A \in \mathcal{M}$, $e = (e_1, \ldots, e_n) \in \tilde{\mathcal{K}}_{N,m}$
   Output: $A'$ with $E_{A'} = (\prod_i \mathfrak{l}_i^{e_i}) \star E_A$
1  $(\mu_1, \ldots, \mu_B) \leftarrow (m_1, \ldots, m_B)$ ;
2  Choose $e^+ \in \mathbb{Z}^n_{\geq 0}$ and $e^- \in \mathbb{Z}^n_{\leq 0}$ s.t. $e_i^+ + e_i^- = e_i$ and
   $\sum_{j=1}^{N_i}(|e^+_{i,j}| + |e^-_{i,j}|) = m_i$ for $1 \leq i \leq n$ ;
3  while $(\mu_1, \ldots, \mu_B) \neq (0, \ldots, 0)$ do
4      Let $I = (I_1, \ldots, I_k)$ s.t. $I_1 < \cdots < I_k$ and
        $\{I_1, \ldots, I_k\} = \{1 \leq i \leq B \mid \mu_i > 0\}$ ;
5      for $1 \leq i \leq k$ do
6          Choose $J_i$ such that $e^+_{I_i,J_i} \neq 0$ or $e^-_{I_i,J_i} \neq 0$ ;
7          $\epsilon_i \leftarrow \begin{cases} 1 & \text{if } e^+_{I_i,J_i} \neq 0 \\ -1 & \text{if } e^+_{I_i,J_i} = 0 \end{cases}$
8      $(A, f) \leftarrow \beta_{R,I}(A, (\epsilon_1, \ldots, \epsilon_k), J)$ ;  // Restricted square-free
        AB
9      for $1 \leq i \leq k$ do
10         $\mu_{I_i} \leftarrow \mu_{I_i} - f_i$ ;
11         if $\epsilon_i = 1$ then
12             $e^+_{I_i,J_i} \leftarrow e^+_{I_i,J_i} - \epsilon_i \cdot f_i$
13         else
14             $e^-_{I_i,J_i} \leftarrow e^-_{I_i,J_i} - \epsilon_i \cdot f_i$
15 return $A$
```

Batching improves dummy-free operation counts even more than it improves constant-time operation counts. However, various subroutines inside ABs need to be redone to avoid lower-level dummy operations or to double-check, preferably at low cost, that the operations are being performed correctly. For example, the constant-time differential addition chains in our software involve dummy differential additions; it should be possible to avoid these by precomputing chains of the same length for all of the primes in a batch. As another example, the Matryoshka-doll structure involves dummy operations, and it would be interesting to explore adaptations of the countermeasures of [40] to this context.

4.2.9.1 Elligator safety

The literature on algorithms for the CSIDH action frequently uses Elligator outputs as cheaper replacements for the uniform random points generated in these algorithms. This appendix analyzes the question of whether this is secure. The conclusion, in a nutshell, is that it seems reasonable to conjecture indistinguishability of the orders of Elligator outputs for large p from the orders of uniform random points.

The cost of UniformRandomPoints. The obvious way to generate a point in $E_A(\mathbb{F}_p)$ is to generate a uniform random $x \in \mathbb{F}_p$ and compute $y = \pm\sqrt{x^3 + Ax^2 + x}$, trying again if $x^3 + Ax^2 + x$ is not a square. The distribution is not exactly uniform, but one can easily adjust the procedure to correct this (see [23, Section 4.1]), or simply accept the distribution as being statistically indistinguishable from uniform.

The standard way to try to compute a square root, given that $p \equiv 3 \pmod 4$, is to compute a $(p+1)/4$ power. One more squaring then reveals whether the input was a square. Generating a point in $E_A(\mathbb{F}_p)$ in this way takes two exponentiations on average.

Before trying to compute y one can check the Legendre symbol $\left(\frac{x^3+Ax^2+x}{p}\right)$. The square-root attempt will succeed if and only if the symbol is not -1. This reduces two exponentiations to two Legendre-symbol computations and one exponentiation, saving time if a Legendre-symbol computation is more than twice as fast as an exponentiation.

Similar comments apply to $\tilde{E}_A(\mathbb{F}_p)$, producing an average `UniformRandomPoints` cost of four exponentiations, or four Legendre-symbol computations and two exponentiations. One can easily reduce the cost to three exponentiations, or two Legendre-symbol computations and two exponentations, by taking the first x as generating a point in $E_A(\mathbb{F}_p)$ in half of the cases and generating a point in $\tilde{E}_A(\mathbb{F}_p)$ in the other half of the cases.

Conventional algorithms for Montgomery-curve computations, including the isogeny computations needed in CSIDH, work only with x and do not need to inspect y. One can thus reduce the cost of `UniformRandomPoints` to three Legendre-symbol computations on average.

Our `high-ctidh` software follows previous CSIDH work in computing a Legendre symbol as a $(p-1)/2$ power, so the speed is the same as computing a square root, but it would be interesting to investigate faster algorithms. One can use blinding to guarantee constant-time Legendre-symbol computation:

- If $x^3 + Ax^2 + x$ is 0, set a bit indicating this, and replace the 0 with 1.
- Multiply by $\pm r^2$ where r is a uniform random nonzero element of $\mathbb{F}_p$.
- Use any Legendre-symbol algorithm.
- Adjust the output according to the 0 bit and the $\pm$ bit.

It would also be interesting to investigate whether the techniques of [24] can be adapted to this context, avoiding the costs of blinding.

Elligators everywhere. The literature generally takes a different approach, using the Elligator 2 [21] map. This approach has the advantage of using just one Legendre-symbol computation to generate a point in $E_A(\mathbb{F}_p)$ and, with no extra cost, a point in $\tilde{E}_A(\mathbb{F}_p)$. The disadvantage is that each point produced is distinguishable from uniform, covering only $(p-3)/2$ out of the $p+1$ possible points. Perhaps the *orders* of these points are distinguishable from the orders of uniform random points.

Elligator was first used in the CSIDH context in [23], which analyzed algorithms to compute CSIDH in superposition as a subroutine inside quantum attacks. That paper mentioned experiments suggesting that Elligator outputs have "failure chance almost exactly $1/\ell$" and that the higher-level algorithms in [23] performed as predicted. However, the security question for constructive CSIDH applications, namely the order-indistinguishability question, did not arise in [23]. A measurable deviation in orders could easily have avoided detection by the experiments in [23].

Elligator was first used for constructive CSIDH applications in [136] to generate an element of $E_A(\mathbb{F}_p)$. It was then used in [154] to generate an element of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$. Subsequent CSIDH software has also used Elligator. It is conceivable, however, that information about A is leaked via the distribution of orders of the points that are generated by Elligator.

Elligator tracking. To directly address the order-distinguishability question, we collected complete data for various small primes p. Specifically, for each $k \in \{1, 2, 3, 4, 5\}$, we took the smallest prime $p \equiv 3 \pmod 8$ for which $(p+1)/4$ factors into exactly k distinct primes.

For each p, we enumerated all $A \in \mathcal{M}$. For each (p, A), we enumerated all Elligator outputs $T_0 \in E_A(\mathbb{F}_p)$ and computed the exact distribution of the order of $[4]T_0$. We then compared this to the uniform model: the exact distribution of orders for uniform random elements of $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. Specifically, we computed the total-variation distance between these two distributions; recall that the total-variation distance between D and E, the conventional form of statistical distance, is $\sum_o |D_o - E_o|/2$.

For example, for $k = 1$ and $p = 11 = 4{\cdot}3-1$, the Elligator order distribution for each $A \in \{0, 5, 6\}$ is 100% order 3: if T_0 is output by Elligator then $[4]T_0$ always has order 3. The uniform model is that orders 3 and 1 appear with probability 2/3 and 1/3 respectively. The total-variation distance is $(|1 - 2/3| + |0 - 1/3|)/2 = 1/3$.

For $k = 2$ and $p = 59 = 4{\cdot}3{\cdot}5-1$, the uniform model is that orders 15, 5, 3, and 1 appear with probability 8/15, 4/15, 2/15, 1/15 respectively. Elligator for $A = 6$ has probability 6/14, 6/14, 2/14, 0 respectively, with total-variation distance $6/35 \approx 0.171429$. Elligator for $A = 11$ has a different distribution from $A = 6$: probability 8/14, 4/14, 1/14, 1/14 respectively. There are 9 choices of A overall, with total-variation distances ranging from ≈0.0619048 to ≈0.171429, averaging ≈0.110582.

Seeing two different values of A with different distributions shows that the result of replacing `UniformRandomPoints` with Elligator is not exactly an atomic block. This does not end the security analysis: for security it is enough to have something indistinguishable from an atomic block. If the total-variation distance drops quickly enough to reach, e.g., 2^{-128} for $p \approx 2^{512}$, then the Elligator orders are indistinguishable from uniform-point orders for every A, and are thus indistinguishable from one A to another.

For $p = 419 = 4 \cdot 3 \cdot 5 \cdot 7 - 1$, there are 27 choices of A, with total-variation distances averaging ≈0.0655745, ranging from ≈0.0357143 to ≈0.119780. For $p = 12011 = 4 \cdot 3 \cdot 7 \cdot 11 \cdot 13 - 1$, there are 195 choices of A, with total-variation distances averaging ≈0.0135444, ranging from ≈0.0063736 to ≈0.0232127. For $p = 78539 = 4{\cdot}3{\cdot}5{\cdot}7{\cdot}11{\cdot}17-1$, there are 459 choices of A, with total-variation distances averaging ≈0.00713331, ranging from ≈0.00353921 to ≈0.0115945. For $p = 1021019 = 4 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 - 1$, there are 1905 choices of A, with total-variation distances averaging ≈0.00493310, ranging from ≈0.00272376 to ≈0.00790233.

To see more information regarding the distributions, we inspected, for each A with $p = 419$, the full distribution of orders of (4 times) Elligator points in $E_A(\mathbb{F}_p)$. The green curves in Figure 4.5 show the minimum, quartiles, and maximum of the per-A distributions; for comparison, the red curve shows the uniform model. Figure 4.6 is for $p = 12011$. The green curves are closer to the red curve for $p = 12011$ than for $p = 419$.

Elligator simulators. Consider the following simulator, resampling from the uniform model: for each $A \in \mathcal{M}$, generate a uniform random sequence of $(p-3)/2$ elements of $[4]E_A(\mathbb{F}_p) \cong \mathbb{Z}/((p+1)/4)\mathbb{Z}$, and compute the distribution of orders of these elements.

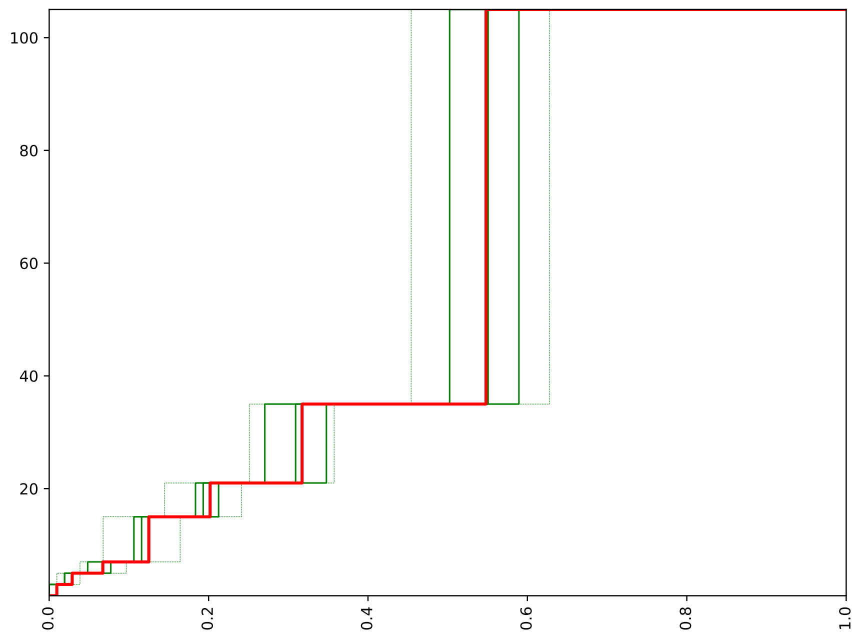


Figure 4.5: For $p = 419$, distributions of orders of points $[4]T_0$ in $E_A(\mathbb{F}_p)$. Red curve: the uniform model, choosing T_0 uniformly at random from $E_A(\mathbb{F}_p)$. Five green curves: minimum, first quartile, median, third quartile, and maximum of the per-A distributions when T_0 is output by Elligator.

Obviously this simulator is not exactly Elligator. For example, Elligator produces each element of $[4]E_A(\mathbb{F}_p)$ at most 4 times. More fundamentally, Elligator deterministically produces a particular distribution for each A, while the simulator produces a new random choice each time. As an extreme case, for $p = 11$, Elligator produces 100% order 3 as noted above, whereas for each $A \in \{0, 5, 6\}$ the simulator produces 100% 3 with probability $16/3^4$; 75% 3 and 25% 1 with probability $32/3^4$; 50% 3 and 50% 1 with probability $24/3^4$; 25% 3 and 75% 1 with probability $8/3^4$; and 100% 1 with probability $1/3^4$.

However, within the range of our experiments, this simulator produces similar results to Elligator. Compare Figure 4.6 to Figure 4.7, which gives an example of the simulator output for $p = 12011$.

A heuristic analysis of this simulator for arbitrary sizes of p proceeds as follows. For each positive integer d dividing $(p+1)/4$, there are $\varphi(d)$ elements of order d in $\mathbb{Z}/((p+1)/4)\mathbb{Z}$, where φ is Euler's phi function. For example, there is 1 element of order 1, and there are $(\ell_1 - 1)\cdots(\ell_n - 1)$ elements of order $(p+1)/4 = \ell_1\cdots\ell_n$. Order d thus occurs with probability $4\varphi(d)/(p+1)$.

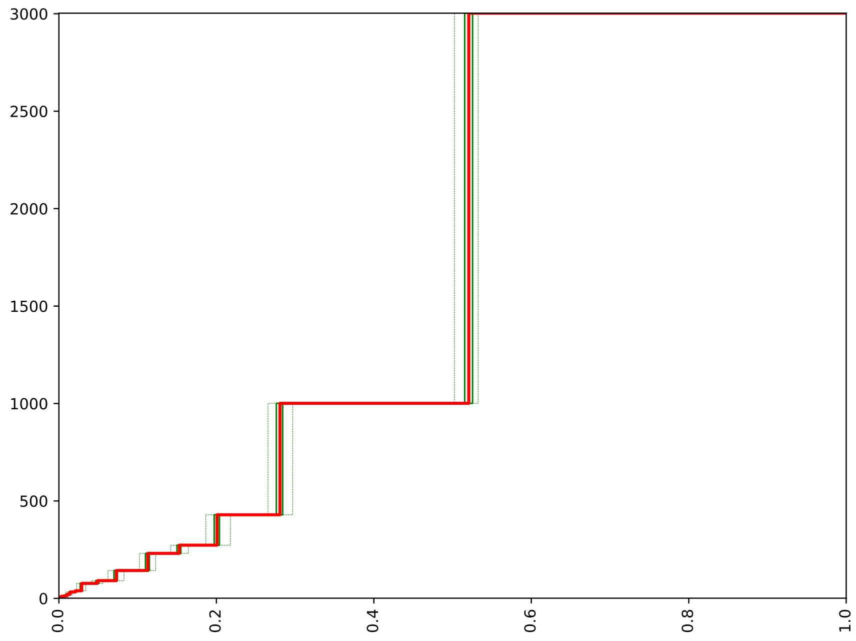


Figure 4.6: Same as Figure 4.6 but for $p = 12011$.

In general, if a trial succeeds with probability q, then the number S of successes in N independent trials has average qN and standard deviation $\sqrt{q(1-q)N}$, so the ratio $(S-qN)/\sqrt{q(1-q)N}$ (assuming $0 < q < 1$) has average 0 and standard deviation 1. The distribution of $(S-qN)/\sqrt{q(1-q)N}$ rapidly approaches a normal distribution as $q(1-q)N$ increases. If the distribution were exactly normal then the half absolute value $|S-qN|/(2\sqrt{q(1-q)N})$ would have average $\sqrt{1/2\pi}$. One thus expects the variation $|S/N-q|/2$ to be approximately $\sqrt{q(1-q)/2N\pi}$ on average.

In particular, write S_d for the number of times that order d occurs among $(p-3)/2$ independent samples from the uniform distribution on $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. Then S_d has average $q_d(p-3)/2$, where $q_d = 4\varphi(d)/(p+1)$, and standard deviation $\sqrt{q_d(1-q_d)(p-3)/2}$. One expects the variation $|2S_d/(p-3)-q_d|$ to be approximately $\sqrt{q_d(1-q_d)/(p-3)\pi}$ on average.

Summing over d says that the total-variation distance is, on average, approximately $\sum_d \sqrt{q_d(1-q_d)/(p-3)\pi}$. This is at most $\sum_d \sqrt{q_d/(p-3)\pi} = X/\sqrt{(p-3)\pi}$ where $X = \sum_d \sqrt{q_d}$. Notice that X factors as $\prod_j(\sqrt{1-1/l_j} + \sqrt{1/l_j})$, which is easy to compute even when p is large. For example, for the CSIDH-512 prime p, this product X is below 2^{11}, while $1/\sqrt{(p-3)\pi}$ is around 2^{-256}.

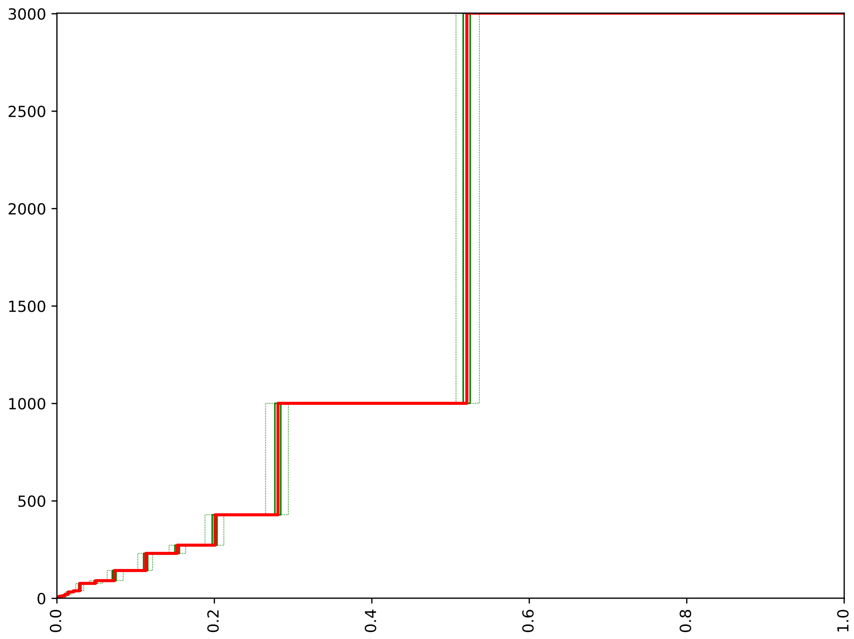


Figure 4.7: Simulation of Figure 4.6. The red curve is copied from Figure 4.6. For the green curves, the Elligator order distributions are replaced by random samples from the red curve.

There are roughly $\sqrt{p}$ choices of A, and they usually vary in total-variation distance. For any particular d, one expects that there exists some A with approximately $\sqrt{\log p}$ standard deviations in the probability that d occurs: e.g., 19 standard deviations for the CSIDH-512 prime p (although finding just 10 deviations would be a large computation). This effect is on a smaller scale than X when $(p+1)/4$ has many small prime factors: i.e., the typical variation accumulated by many large divisors d is larger than occasional variation for the largest divisor. One does not expect several standard deviations to occur for several d simultaneously.

p	$\#\mathcal{E}$	$\lg X$	$\lg H$	$\lg H_1$	min	avg	max				ratios
11	3	0.48	−1.85	−2.41	−1.58	−1.58	−1.58	0.68	1.20	1.20	1.20
59	9	0.90	−2.83	−3.12	−4.01	−3.18	−2.54	0.82	0.44	0.78	1.22
419	27	1.29	−3.89	−4.07	−4.81	−3.93	−3.06	0.89	0.53	0.97	1.78
12011	195	1.50	−6.10	−6.26	−7.29	−6.21	−5.43	0.90	0.44	0.93	1.60
78539	459	1.89	−7.06	−7.16	−8.14	−7.13	−6.43	0.94	0.47	0.95	1.55
1021019	1905	2.20	−8.61	−8.67	−9.52	−8.66	−7.98	0.96	0.53	0.96	1.54

Table 4.5: Heuristic analysis of the Elligator simulator (see text) compared to actual Elligator order distributions. “$\#\mathcal{E}$”: number of choices of A. “$\lg X$”: logarithm base 2 of $X = \prod_j(\sqrt{1-1/l_j} + \sqrt{1/l_j})$. “$\lg H$”: logarithm base 2 of $H = \sum_d \sqrt{q_d/(p-3)\pi} = X/\sqrt{(p-3)\pi}$. “$\lg H_1$”: logarithm base 2 of $H_1 = \sum_d \sqrt{q_d(1-q_d)/(p-3)\pi}$. “min” and “avg” and “max”: logarithm base 2 of the minimum and average and maximum, over A, of the total-variation distance between Elligator and the uniform model. “ratios”: H_1 and minimum and average and maximum, divided by H, without logarithms.

To summarize, one expects all total-variation distances from the simulator to be close to $X/\sqrt{(p-3)\pi}$. The ratios in Table 4.5 show that the actual Elligator total-variation distances are close to $X/\sqrt{(p-3)\pi}$ for $p \in \{11, 59, 419, 12011, 78539, 1021019\}$.

Zero hazards. The original Elligator paper [21] did not define Elligator 2 for $A = 0$. The application of Elligator to CSIDH attacks in [23] suggested handling $A = 0$ by precomputing a point of full order, or, alternatively, replacing the initial $A/(r^2 - 1)$ with r when $A = 0$.

Our CTIDH software (see Section 4.2.7) replaces the initial $A/(r^2 - 1)$ with $1/(r^2 - 1)$ when $A = 0$. For constant-time projective computations this seems slightly more efficient than replacing $A/(r^2 - 1)$ with r. We included this handling of $A = 0$ in the computations described above of the total-variation distance.

Beware that the alternative of precomputing a point of full order would not generally be safe in constructive applications. This precomputation eliminates failure cases for $A = 0$, making $A = 0$ easily distinguishable from other values of A via timing. An attacker that guesses the isogenies used in the victim's first AB, and provides a fake public key that is taken to 0 by those isogenies, can check this guess by watching timings of the victim's second AB. Once the attacker has enough confidence regarding the first isogenies, the attacker can move on to guessing the isogenies used in the victim's second AB. The attacker continues in this way to adaptively target the victim's full private key. This adaptive timing attack breaks [136, Section 5.3, second paragraph]. On the other hand, always using a large enough number of iterations to reach a negligible failure probability, as in [23], would stop this attack.

One Elligator, or two? The order-distinguishability question for orders in $\tilde{E}_A(\mathbb{F}_p)$ is equivalent to the order-distinguishability question for orders in $E_{-A}(\mathbb{F}_p)$, so it does not need to be analyzed separately. However, a different security question arises if a single `UniformRandomPoints` call is replaced with a single Elligator call, as in [154] and subsequent constant-time CSIDH work. It is conceivable, for example, that the resulting element of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$ has a measurable correlation between 3 dividing the order in the $E_A(\mathbb{F}_p)$ part and 5 dividing the order in the $\tilde{E}_A(\mathbb{F}_p)$ part, even if the order in each part separately is indistinguishable from uniform.

Our `high-ctidh` software simplifies the security analysis by using Elligator once to generate an element of $E_A(\mathbb{F}_p)$, and using Elligator again to generate an independent element of $\tilde{E}_A(\mathbb{F}_p)$. It is not clear that this is a slowdown: independently generating two points lets the software save the cost of pushing one of the two points through an isogeny, and Legendre-symbol computations could be fast enough to justify this purely from a speed perspective. (Note that if Legendre-symbol computations have low enough cost then it is easy to argue for incurring the slowdown of using two Legendre-symbol computations on each curve to generate uniform random points, skipping Elligator and further simplifying the security analysis.)

On the other hand, a single Elligator call could be best for speed, and is used in several previous papers. So we also studied the joint distribution of $E_A(\mathbb{F}_p) \times \tilde{E}_A(\mathbb{F}_p)$ orders.

For $A = 0$, the Elligator extensions mentioned above produce outputs of the form $((x, y), (-x, iy))$. The "distortion map" from (x, y) to $(-x, iy)$ is compatible with elliptic-curve addition, so it preserves the order of points. This reduces the security question for $E_0(\mathbb{F}_p) \times \tilde{E}_0(\mathbb{F}_p)$ to the security question for $E_0(\mathbb{F}_p)$, which was addressed above.

For nonzero A, our computations did not detect any such correlations. We instead compared the pair of orders to a uniform-pair model, namely the orders of two independent uniform random elements T_0, T_1 of $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. We found total-variation distance averaging ≈ 0.312619 (maximum ≈ 0.358730) for $p = 59$, ≈ 0.207722 (maximum ≈ 0.260199) for $p = 419$, ≈ 0.0512755 (maximum ≈ 0.0678949) for $p = 12011$, and ≈ 0.0361776 (maximum ≈ 0.0416506) for $p = 78539$.

It is not surprising that the distance from the joint distribution to the uniform-pair model is generally larger than the distance from the single-point distribution to the uniform model. There are many more possibilities for a pair of orders than for a single order.

To quantify this, consider a joint-distribution simulator that resamples from the uniform-pair model. If d_1 and d_2 are positive integers dividing $(p+1)/4$ then there are $\varphi(d_1)\varphi(d_2)$ pairs of elements T_1, T_2 of orders d_1, d_2 respectively in $\mathbb{Z}/((p+1)/4)\mathbb{Z}$. A heuristic analysis proceeds as before, with q_d replaced by $q_{d_1} q_{d_2}$, and X replaced by X^2, giving the estimate $X^2/\sqrt{(p-3)\pi}$. This estimate is ≈ 0.263654 for $p = 59$, ≈ 0.164434 for $p = 419$, ≈ 0.0410512 for $p = 12011$, and ≈ 0.0277182 for $p = 78539$. If the actual joint-Elligator distances remain close to $X^2/\sqrt{(p-3)\pi}$ for all CSIDH primes p then the distances are acceptably small for CSIDH-512.

5 Physical Attacks

5.1 Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations against Fault Injection Attacks

This chapter is for all practical purposes identical to the paper *Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations against Fault Injection Attacks* [40] authored jointly with Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger, which was published at FDTC 2020.

5.1.1 Introduction

Isogeny-based cryptography is a promising candidate for quantum-resistant schemes. The most popular schemes, SIDH (Supersingular Isogeny Diffie–Hellman) and CSIDH (Commutative Supersingular Isogeny Diffie–Hellman), offer key-exchange protocols with the smallest key sizes, but the worst performance among all current post-quantum schemes. In contrast to SIDH, CSIDH is non-interactive, and basically could be used as a drop-in replacement for current applications of Diffie–Hellman or ECDH. However, it seems that until now, isogeny-based PQC schemes have not received much attention in the implementation-attack literature [58]. Only few fault-injection attacks on SIDH and more general investigations ([94, 180, 91]) have been discussed and published in the community so far. In [51], fault-injection attacks on a constant-time implementation of CSIDH have been discussed. However, all previous publications only consider attacks on a theoretical level and omit discussing a particular fault model, fault-attack method, and fault-injection technique. To the best of the authors' knowledge in none of the publications the practical execution of fault-injection attacks has been investigated. Therefore, this is the first work on practical evaluation on the feasibility of fault attacks on an implementation of the CSIDH key-exchange protocol.

In this work we focus on CSIDH, for which there are currently two proposals to design constant-time implementations. One approach uses dummy computations to achieve time constantness ([136, 154, 51]), while the other is dummy-free ([51]). The former approach is believed to be less secure against fault attacks, but is twice as fast as the latter. In this work, we evaluate practical fault attacks on the former approach, and present countermeasures, leading to a relatively small slowdown by a factor of 1.07, which yields a significantly better performance than the dummy-free alternative.

The contributions of this work are as follows: First, we discuss practical attacker models for fault attacks and side-channel assisted fault attacks on constant-time CSIDH implementation with dummy isogenies. We then simulate all discussed attack models and perform practical experiments with low-budget attack equipment. Lastly, we evaluate the performance of the proposed countermeasures in practice.

We place the code used for this work into the public domain; it is available at `https://github.com/csidhfi/csidhfi` and `https://doi.org/10.5281/zenodo.6900027`. It includes the CSIDH implementation with and without countermeasures, the attack-simulation scripts, and attack scripts.

Remark 1. The majority of this work was done prior to the publication of asymptotically faster isogeny formulas by Bernstein, De Feo, Leroux, and Smith [19]. Some of our countermeasures rely on the structure of the isogeny computations in the implementations [51, 136, 154]. Since this is significantly altered in the formulas from [19], it is unclear whether they can be protected by similar countermeasures. However, for small degrees the formulas used in this work are still faster, and it is yet unclear for which threshold the new formulas become faster in a constant-time implementation. Even if there are no similar countermeasures for [19], one could design a hybrid implementation, where the small degrees use protected dummy computations, while the larger degrees use the dummy-free approach.

5.1.2 Preliminaries

5.1.2.1 CSIDH

We focus on an algorithmic description of CSIDH here; for more background, we refer to [49].

First, we define a prime of the form $p = 4\ell_1 \cdots \ell_n - 1$, where $\ell_1, \dots, \ell_n$ are small distinct odd primes, and work with supersingular elliptic curves in Montgomery form $E_A : y^2 = x^3 + Ax^2 + x$ over $\mathbb{F}$. Therefore, each such curve contains points of orders ℓ_i for all $1 \leq i \leq n$, which can be used as input to compute an isogeny of degree ℓ_i, e.g. using the formulas of [62]. A private key is given by a vector of integers $(e_1, \dots, e_n)$, where the entry e_i determines that $|e_i|$ isogenies of degree ℓ_i have to be computed, and the sign of e_i determines if an order-ℓ_i point on the current curve or its twist has to be taken as the input. The entries are sampled from a small interval $[-m, m]$ to obtain an efficient computation. This so-called class group action evaluation thus takes as input a curve E, computes the required chain of isogenies, and outputs a different curve E'. Note that the order of computing the required isogenies is not fixed, due to the commutativity of this action. In practice, efficient algorithms for this class group action sample a point on the current curve, and compute as many isogenies from this point as possible, thereby requiring to push this point through each computed isogeny in this chain. This can be seen in Algorithm 2 of [49].

The commutativity immediately allows us to set up a Diffie–Hellman-style key exchange: Alice and Bob agree on an initial curve E_0 and choose private-key vectors. Both compute the respective class group action, and obtain a public key E_A resp. E_B. Then, Alice repeats the computation of her chain of isogenies with the starting curve E_B, and Bob proceeds vice versa with E_A. Because of the commutativity, both parties then arrive at the same curve E_{AB}, which can be used as the shared secret. This key exchange is non-interactive, and due to the efficient verification of public keys, allows for static-static key exchange [49].

5.1.2.2 Isogenies

As shown by Costello and Hisil [62], for curves in Montgomery form, an isogeny $\varphi : E \to E'$ of odd degree $\ell = 2d + 1$ can be computed from the following formulas. Let $K \in E$ be a point of order ℓ, and denote by $(X_i : Z_i)$ the projective coordinates of the point $[i]K$. Then

$$\begin{aligned}\varphi : (X : Z) \mapsto \\ &\left(X \left(\prod_{i=1}^{d} (X - Z)(X_i + Z_i) + (X + Z)(X_i - Z_i) \right)^2 : \right. \\ &\qquad \left. Z \left(\prod_{i=1}^{d} (X - Z)(X_i + Z_i) - (X + Z)(X_i - Z_i) \right)^2 \right). \quad (5.1)\end{aligned}$$

The curve parameter $a' = (A' : C')$ of E' can be computed by formulas by Meyer and Reith [137], exploiting the birational equivalence to a twisted Edwards curve:

$$\begin{aligned}(A' : C') = \big(2 \cdot \big((A + 2)^{\ell} \pi_+^8 + (A - 2)^{\ell} \pi_-^8\big) : \\ (A + 2)^{\ell} \pi_+^8 - (A - 2)^{\ell} \pi_-^8\big), \quad (5.2)\end{aligned}$$

where

$$\pi_+ = \prod_{i=1}^{d}(X_i + Z_i) \quad \text{and} \quad \pi_- = \prod_{i=1}^{d}(X_i - Z_i).$$

Dummy isogenies

As suggested by Meyer and Reith [137], constant-time algorithms of CSIDH often use dummy isogenies, since otherwise the running time is correlated to the secret key, which specifies the number of isogenies to be computed. These dummy computations perform the same instructions as real isogeny computations, but discard the results. Thus, they allow for a fixed number of isogeny computations, independent from the respective private key.

In order to speed up computations, dummy isogenies are designed to compute $[\ell]P$ for the input point P. This has to be done, since for a real isogeny of degree ℓ, the order of P loses the factor ℓ by being pushed through. Therefore, a dummy isogeny would require a subsequent multiplication $[\ell]P$, which is prevented by performing this computation inside the dummy algorithm. To this end, a dummy isogeny swaps the input points K (kernel point) and P (point to be evaluated), to compute $[(\ell-1)/2]P$ in the kernel computation part. Then two further differential additions suffice to compute $[\ell]P$. However, this method requires to perform these two further additions in a real isogeny as well, and discard their results, in order to achieve a constant-time behavior.

Figure 5.1 and Figure 5.2 show the different computation blocks that are contained in the degree-ℓ isogeny algorithm. For real and dummy isogenies, the green blocks are necessary computations in order to produce a valid output, while the red blocks entirely consist of dummy computations, whose results are discarded. Note that these figures do not show conditional swaps, which are necessary to avoid conditional branches based on the private key. We refer to [136] and the accompanying implementation for more details.

5.1.2.3 Constant-time algorithms

Meyer, Campos, and Reith (MCR) [136] pointed out that in addition to the variable number of isogenies, also the sign distribution of the key elements may leak information through the running time. Thus, they proposed a constant-time algorithm of CSIDH by using dummy isogenies, and by changing the secret key intervals from $[-m, m]^n$ to $[0, 2m]^n$. As a result, for any secret key the performance is the same as for the action of the integer vector $(2m, \ldots, 2m)$. This cost is about twice as much as that of the action of $(m, \ldots, m)$, which is the worst case in the variable-time algorithm. Further, they proposed several optimizations, such as the batching technique SIMBA or the usage of the point sampling method Elligator [21], which was first used in the context of CSIDH in [23], and obtain a speed-up factor of roughly 2.

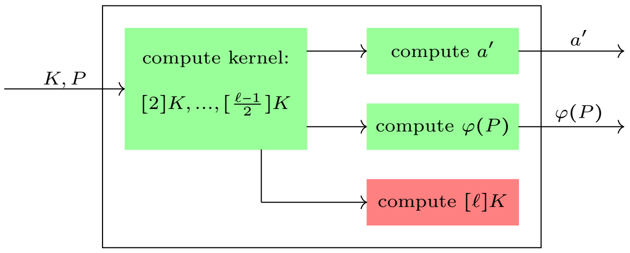


Figure 5.1: Real isogeny

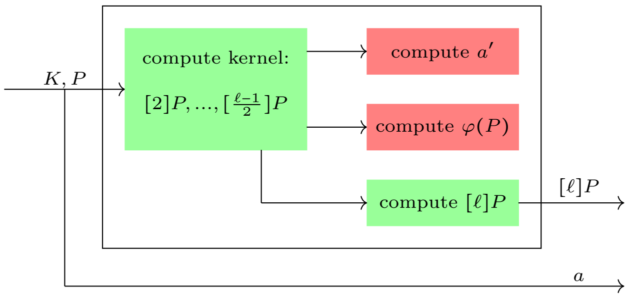


Figure 5.2: Dummy isogeny

Onuki, Aikawa, Yamazaki, and Takagi (OAYT) [154] proposed an idea for mitigating the increase of the computational cost due to the key interval $[0, 2m]$. By keeping two points $P_0 \in E[\pi - 1]$ and $P_1 \in E[\pi + 1]$ in each step in the algorithm, where π denotes the Frobenius endomorphism, one can compute isogenies for positive signs and negative signs of a secret key in the same loop. By always choosing the point P_s that suits the sign of e_i for computing the kernel generator of an ℓ_i-isogeny, the correlation between running time and sign distribution is eliminated. Thus, this method allows for the use of the secret key intervals $[-m, m]^n$, and therefore halves the number of total isogenies at the cost of an additional point evaluation per isogeny. We describe their algorithm in Algorithm 13. Note that, for the sake of simplicity, optimizations such as SIMBA are not described in Algorithm 13. We refer to [136, 154] for more details.

Cervantes-Vázquez, Chenu, Chi-Domínguez, De Feo, Rodríguez-Henríquez, and Smith (CCCDRS) [51] obtained a speedup for the MCR and OAYT implementations by using twisted Edwards curves. Further, they proposed a dummy-less implementation in order to improve the resistance against fault attacks, at the cost of a slowdown by a factor of 2.

Algorithm 13: Constant-time class group action

```
1  Set $e'_i = m - |e_i|$ for $i = 1, \dots, n$.
2  while some $e_i \neq 0$ or $e'_i \neq 0$ do
3      Set $S = \{i \mid e_i \neq 0 \text{ or } e'_i \neq 0\}$.
4      Set $k = \prod_{i \in S} \ell_i$.
5      Generate $P_0 \in E_A[\pi - 1]$ and $P_1 \in E_A[\pi + 1]$ by Elligator.
6      Let $P_0 \leftarrow [(p+1)/k]P_0$ and $P_1 \leftarrow [(p+1)/k]P_1$.
7      for $i \in S$ do
8          Set $s$ the sign bit of $e_i$.
9          Set $K = [k/\ell_i]P_s$.
10         Let $P_{1-s} \leftarrow [\ell_i]P_{1-s}$.
11         if $K \neq \infty$ then
12             if $e_i \neq 0$ then
13                 Compute $\varphi : E_A \to E_B$ with $\ker\varphi = \langle K \rangle$.
14                 Let $A \leftarrow B$, $P_0 \leftarrow \varphi(P_0)$, $P_1 \leftarrow \varphi(P_1)$, and
                    $e_i \leftarrow e_i - 1 + 2s$.
15             else
16                 Compute dummy isogeny:
17                 Let $A \leftarrow A$, $P_s \leftarrow [\ell_i]P_s$, and $e'_i \leftarrow e'_i - 1$.
18         Let $k \leftarrow k/\ell_i$.
19 return $A$.
```

5.1.3 Attacker Models

The attacker we are modeling in this work is deploying safe-error analysis to detect the dummy isogenies within CSIDH, i.e., he injects faults during the computation of the CSIDH group action and observes if an occurring fault impacts the shared secret. An adversary that can reliably skip or corrupt an isogeny computation of a chosen degree at a chosen index can easily recover the full secret key with a relatively small number of fault injections. However, due to various sources of randomness during the execution, it is impossible to always corrupt the intended operation and without side-channel information an adversary cannot know which isogeny was affected. Therefore, we propose three different attacker models with increasing capabilities to evaluate the impact of the resulting attacks.

In general, we assume that an adversary is able to repeatedly trigger an evaluation of the group action using the same secret key. The input curve may be the same for all evaluations, but may also be different. As CSIDH allows a static-static key exchange, this is likely how a key exchange is implemented. The attacker is able to inject faults that will set variables to random values or skip instructions. An attacker is limited to observe whether both parties obtained the same shared secret, e.g., by observing failure later in the protocol. Expressed in a more formal way, this model is the same as the second oracle from [91]. We propose the following three attackers with increasing capabilities. Attacker 1 and Attacker 2 are limited to fault injection, while Attacker 3 can also obtain additional side-channel information.

- **Attacker 1: Shotgun at the CSIDH.** Our weakest adversary model assumes that the attacker can reliably cause a fault during the computation of the CSIDH group action, but has no control over the location of the fault. He can then observe how often this leads to a wrong shared secret. This proportion of failures intuitively is depending on the ratio of "real" vs. "dummy" isogenies. While this is a rather weak adversary model, it nicely demonstrates the inherent problem of dummy operations in the context of fault injection attacks.

  The main limitation of Attacker 1 is that he has no control over the operation that is affected. Since the isogeny computations make up about 42% of cycles during the group action on the Cortex-M4, the attacker is likely to hit an isogeny computation relatively often. However, he has no knowledge of the order of the faulty isogeny computation which limits the information he can learn about the secret key.

- **Attacker 2: Aiming at isogenies at index i.** A slightly more powerful adversary can target isogeny computations at positions of his choice. This does not fully allow to target isogenies of a chosen degree, as the isogenies may be evaluated out of order due to point rejections. However, since the first evaluated isogenies have relatively large orders ℓ_i, and the point rejection probability is $1/\ell_i$, the sequence of the first isogenies is almost deterministic and the individual isogenies can be targeted easily. We evaluate how many isogenies the adversary can realistically attack in Section 5.1.4.

  For all entries of the secret key with $e_i = 0$, the injected fault will not change the result, and an adversary immediately knows this part of the secret key. For the remaining e_i the adversary has reduced the search space.

- **Attacker 3: Aiming at isogeny computations and tracing the order.** Our most powerful attacker model complements attacker 2 by additionally allowing the adversary to trace the faulty isogeny computation to determine the degree of the isogeny that the fault was injected into. Since the isogeny order determines the run-time of the isogeny computation the order might be recovered from a power trace, e.g., using Simple Power Analysis [119].

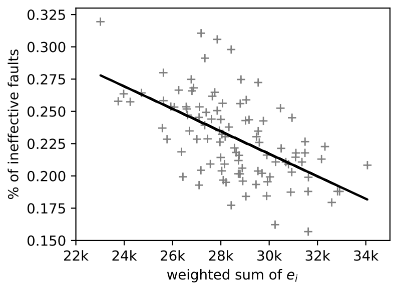


Figure 5.3: Simulation results for Attack 1 using 100 random CSIDH512 secret keys. 500,000 faults are injected into random operations during the group action.

One could imagine yet another adversary that is capable of setting certain e_i of the secret key to a chosen value. A possible attack would be as follows: For each e_i try all possible values and observe for which value the derived shared secret is correct. For the CSIDH512 parameters proposed in [154], this would require at most 882 successful fault injections for fully recovering the secret key. This attack would also apply to dummy-free implementations like [51]. Note, however, that this adversary is overly powerful especially when assuming the low-cost fault injection equipment we are targeting in this work. Therefore, we focus on more realistic fault models for the remainder of this chapter which can be achieved using relatively cheap clock-glitching equipment.

5.1.4 Simulation

To gain a better understanding of how many fault injections an adversary would require to obtain a certain key space reduction or key recovery, we simulate the three previously defined adversary models and mount practical experiments on them.

5.1.4.1 Attack 1

For the simulation Attack 1, we implemented a Python script which simulates all operations that are performed within CSIDH in the OAYT implementation. Our approach works as follows: We use our implemented cost-simulation to output a transcript of each point multiplication, isogeny computation, etc., in addition to their cost. We then select one of these operations using the strategy corresponding to the attack model (e.g., uniformly random for Attack 1) and determine the impact of a fault occurring at that position. This script is parameterized by the relative cost of each operation which we experimentally determine for our target implementation.

In general there can be two outcomes of a simulated fault injection:

- A fault was injected into an operation that was not a dummy operation which will lead to a wrong shared secret in most cases, which can be observed by the adversary.
- A fault was injected into a dummy operation, i.e., there is no change in the shared secret. This can be considered an ineffective fault.

Note that there are some special cases, where a fault was injected into a non-dummy operation, but the resulting shared secret is not influenced by this. Although these cases are rather rare, our simulation still considers them, in order to give more realistic results.

In Attack 1, the adversary simply observes the percentage of fault injections that yield a wrong shared secret. This proportion depends on the secret key as it determines the proportion of real versus dummy operations.

Results We simulated the attack for 100 randomly selected CSIDH512 keys and performed 500,000 fault injections at random locations during the entire group action. Fig. 5.3 shows the plot of the probability of an ineffective fault in relation to the weighted sum of the secret key. As the the runtime of an individual isogeny is linear in its degree, the time spent in ℓ_i-isogenies is proportional to $|e_i| \cdot \ell_i$. Therefore, we compute the weighted sum as $\sum |e_i|\ell_i$ which corresponds to the approximate time spent in real isogenies. From the simulation, it is easy to see that the probability of seeing a faulty shared secret is correlated with the secret key. An adversary learning this probability, can also infer information about the secret key. The more faults are injected, the more evident this relationship becomes.

Impact After obtaining the percentage of ineffective faults for a large enough number of rounds, the attacker now wants to gain information on the used secret key. However, we provide an example to show that this does not lead to a large reduction of the possible key space. Suppose the attacker obtains a percentage that allows him to assume that the weighted sum of the secret key is less than 24k. Then, by a Monte Carlo method, we can estimate that roughly 1% of all the possible CSIDH512 keys satisfy this condition. This means that the search space got reduced from 2^{256} to roughly 2^{249}. Since the correlation between the obtained percentage and weighted sum of the key is not even strong enough to allow for an assumption as in this example, we conclude that this attack is not able to significantly reduce the respective search space.

5.1.4.2 Attack 2

In the proposed constant-time implementations based on dummy isogenies (MCR and OAYT), the calculation for a certain e_i from the secret key vector $(e_1, \ldots, e_n)$ acts deterministically. This means that first real and then dummy isogenies are calculated (see lines 12 - 17 in Algorithm 13). Thus, it is sufficient to determine within this calculation sequence where the first dummy isogeny occurs in order to know the absolute value of each e_i.

We assume in Attack 2 and Attack 3 that the attacker knows spots in the isogeny computation for the respective degree which reveal whether it is a real or dummy isogeny calculation with a single fault injection. Such critical spots (according to Figure 5.1 and Figure 5.2) in the code can be empirically determined in advance with manageable effort. In our experiments, we achieved an accuracy of over 95% with a single fault injection.

Results For the second attack, it suffices to simply determine up to which isogeny computation the algorithm is likely to be deterministic, i.e., no points are going to be re-sampled. Since the probability of point rejection for a given degree ℓ_i is $1/\ell_i$, the sequence of the first isogeny computations is deterministic with high probability, due to the relatively large degrees. For example, the attacker knows with a probability of 71% that the first 23 isogeny computations run without point rejection in the OAYT implementation. This makes it easy to target these first 23 isogenies and find out whether they are real or dummy computations with relatively few fault attempts. Extending this number of 23 isogenies leads to a quickly increasing probability for point rejections, thus preventing unambiguous results for later isogenies.

Impact The space reduction achieved in this attack model is from 2^{256} to 2^{177} in the best case, where all the respective key elements are 0, and roughly to 2^{244} in the average case. For the average case, we assume that 1/11 of the respective key elements, which lie in the range $[-5, 5]$ resp. $[0, 10]$, are 0. In the worst case, i.e. none of the respective key elements being 0, the key space is reduced to 2^{253}

5.1.4.3 Attack 3

Since in this attack model the attacker is also able to trace the order of the isogeny calculation, a divide-and-conquer approach provides the most effective strategy.

Results Both constant-time implementations (MCR and OAYT) use a bound vector $\mathtt{m} = (m_1, m_2, \ldots, m_n)$ defining the intervals from which each secret exponent e_i must be sampled. The number of fault injections required to obtain the absolute value of a certain e_i depends on the corresponding m_i

from the bound vector and on the number of attempts needed to distinguish a real from a dummy isogeny. For each individual degree, the attacker simply performs a binary search until the calculation of the first dummy isogeny is identified.

Impact In the case of the MCR implementation, where only positive values were used for the secret key vector ($e_i \in [0, 2m]$, where $m = 5$), at least 178 injections are required in the worst case for a full key recovery. Whereas in the case of the OAYT implementation our strategy requiring at least 178 injections leads to a space reduction to 2^{74} in the worst case and to 2^{67} in the average case. The remaining search complexity can be further reduced to roughly 2^{38} in the worst case resp. $2^{34.5}$ in the average case by a meet-in-the-middle approach as described in [49].

5.1.5 Practical Experiments

All our fault-injection experiments were performed on a ChipWhisperer-Lite (CW1173) 32-bit basic board, which includes a 32-bit STM32F303 ARM Cortex-M4 processor as the target core. The attacks were implemented in Python (version 3.6.9) using the ChipWhisperer open source toolchain[19] (version 5.1.3). An ARM plain C implementation of CSIDH, based on the implementation by Onuki, Aikawa, Yamazaki, and Takagi (OAYT) [154], was implemented for our project.

To reduce the time required for all experiments on the target board, we reduced the key space from 11^{74} to 3^2, i.e., our secret consists of two elements in $\{-1, 0, 1\}$. Furthermore, in Attack 1 we compute isogenies with the smallest degrees (3 and 5).

In all implemented attacks, the isogenies are calculated without randomness, i.e., points and private keys used were precomputed. To require only one CSIDH action call per experiment, Bob's public key and the resulting shared secret for Alice's given public key were calculated in advance. Specifically, in all the implemented scenarios Alice's computation of the shared secret is attacked.

In our setup, the fault is injected by suddenly increasing the clock frequency, hence, forcing the target core to skip an instruction.

Table 5.1 shows the results for the practical attacks. While the rate for Attack 1 increases slightly for keys containing more real isogenies, the increase for random-based (without knowledge of critical points) Attack 2 is much higher. The results from Attack 2 also apply to Attack 3.

[19] `https://github.com/newaetech/chipwhisperer`, commit 887e6c7

Table 5.1: Results for Attack 1 and Attack 2

type	key	# of trials	faulty shared secret
Attack 1	{0,0}	5000	19.8%
	{0,1}	5000	27.3%
	{-1,1}	5000	32.8%
Attack 2	{0,1}	5000	2.1%
	{-1,1}	5000	16.4%

5.1.6 Countermeasures

We describe countermeasures for the OAYT implementation, but note that this also applies to MCR, and, with slight modifications, to the CCCDRS implementation containing dummy computations.

It is evident that Attack 3 is the main threat that should be considered for countermeasures. Thus, we analyze the required countermeasures for the involved dummy computations during isogenies. However, the simulation of Attack 1 shows that there are other parts of the CSIDH algorithm that could leak some information on the private key, if specifically attacked as in the Attacker 3 model. Therefore, we describe the further required countermeasures, such that the resulting implementation is secure against leakage in all three attack models.

A rather simple countermeasure would be to randomize the order in which real and dummy isogenies for a specific degree are computed, instead of always computing the real ones first. However, Attacker 3 can still attack this with a slightly larger number of faults, using a probabilistic method to obtain the key elements. In contrast to this, our idea for countermeasures against the described fault injection attacks is to redesign the algorithm such that any fault injection will lead to the output of an error instead of the output curve. This means that an attacker does no longer see if the injected fault affected a real or a dummy operation, and is thus effective against all three attack models we described.

A key function that is frequently used is a check for equality. This is performed in constant time and therefore does not leak any information. The presented countermeasures are designed for our described specific attack model, i.e. the adversary is limited to injecting exactly one fault, which can either be a random fault or an instruction skip.

5.1.6.1 Isogenies

In order to reach security against Attack 3, we have to be able to detect faults during the dummy computations of isogenies. However, we stress that we require a unified isogeny algorithm, which computes a real isogeny or dummy isogeny of given degree in constant time, based on a decision bit

Algorithm 14: Protecting the codomain curve

```
Input  : Curve parameters $A, C \in \mathbb{F}$, degree $\ell$, kernel points
         $(X_i : Z_i)$ for $1 \leq i \leq (\ell-1)/2$, bitmask $b \in \{0,1\}$.
Output: Curve parameters $A', C' \in \mathbb{F}$, error variable $error$.
1  Set $\pi_+ \leftarrow 1$, $\pi_- \leftarrow 1$
2  for $i \in \{1,\dots,(\ell-1)/2\}$ do
3      $t_0 \leftarrow$ cadd$(X_i, Z_i, b)$.
4      $t_1 \leftarrow$ csub$(X_i, Z_i, b)$.
5      $\pi_+ \leftarrow \pi_+ \cdot t_0$.
6      $\pi_- \leftarrow \pi_- \cdot t_1$.
7  $t_0 \leftarrow$ cadd2$(C, C, b)$.
8  $t_1 \leftarrow (A - t_0)^\ell \cdot \pi_-^8$.
9  $t_0 \leftarrow (A + t_0)^\ell \cdot \pi_+^8$.
10 $A' \leftarrow$ cadd$(t_1, t_0, b)$.
11 $A' \leftarrow$ cadd$(A', A', b)$.
12 $C' \leftarrow$ csub$(t_0, t_1, b)$.
13 $error \leftarrow$ cverify$(A', C', \neg b)$.
14 return $A', C', error$.
```

$b \in \{0,1\}$. This means that countermeasures for one of the two cases must be executed in *both* cases to maintain the constant-time property. However, it must be ensured that the verifications only lead to the output of an error in the relevant case. This is implemented via the function `cverify`(x, y, b), which always checks whether $x = y$ via the constant-time check for equality, but only outputs the result if $b = 1$.

In this section, we assume that the decision bit is set as $b = 0$ if a dummy isogeny is to be computed, and $b = 1$ for the real isogeny case.

5.1.6.2 Real isogenies

As depicted in Section 5.1.2.2, the two additional differential additions (DADDs) are the only dummy computations in a real isogeny. Since their output is discarded, we have to validate that no fault has been injected during their execution. However, in this case the validation is straightforward. The DADDs are designed to compute $K' = [\ell]K$ for an ℓ-isogeny (see Section 5.1.2.2). Thus, in real isogenies, the result must be the point ∞, since K has order ℓ. This means that we can simply call `cverify`(K', ∞, b), in order to perform this validation only in the case of real isogenies.

5.1.6.3 Dummy isogenies

In dummy isogenies, the dummy computations are the codomain curve computation and the point evaluation, as described in Section 5.1.2.2. However, the involved dummy computations here don't allow for an elegant verification of point orders or supersingularity, as in all the other cases in this section. Instead, we will make use of a conditional addition $\texttt{cadd}(x, y, b)$, which outputs x if $b = 0$ and $x + y$ if $b = 1$. This function is implemented by first calling a conditional set function, which takes as input y and b, is initialized by the output value 0, and overwrites this output by y if $b = 1$. Note that this function is implemented to run in constant time, in order to prevent leakage. Then, we call the usual addition function for $\mathbb{F}$-elements, and obtain the desired output in constant time.

While we have to maintain the structure of computations in the case of real isogenies (i.e. for $b = 1$), we have to make changes to them in order to obtain verifiable results in the dummy case (i.e. for $b = 0$). To manage this in constant-time, we make use of the conditional add function. Analogously, we define a conditional subtraction $\texttt{csub}(x, y, b)$, and can compute $\texttt{cadd2}(x, y, b)$ with result $bx + by$ through two calls to the conditional set function.

Codomain curve computation Instead of using multiples of the kernel generator K, dummy isogenies use multiples of the input point P_s. Thus, the output does not refer to a special type of curve or follow any other special property that can be validated.

Recall that the codomain curve parameters are computed by Eq. 5.2. It is evident that different steps during the computations of A' and C' contain similar terms, and mostly differ in sign changes. Therefore, our strategy to evaluate these computations in the dummy case is to manipulate some of them with conditional additions, in order to obtain $A' = C'$. To reach this, our algorithm is designed in a way such that a fault injection in any line of code leads to $A' \neq C'$ in the dummy case. On the other hand, it obviously computes the correct output parameters in the case of real isogenies. Algorithm 14 details this method. Again we make use of the conditional verify function, to only possibly raise an error if $b = 0$.

Point evaluation Analogously to the codomain curve computation, there is no possibility to check for the correct executions of this part through point order checks in the dummy case. Thus, we resort to the same strategy as for the codomain curve computation.

The output points are computed by Eq. 5.1. We can again use the same strategy to manipulate the computations to output values satisfying $X' = Z'$ in the dummy case. As above, a fault to any line of code will result in output values with $X' \neq Z'$, and in the real isogeny case, the algorithm stays

Algorithm 15: Protecting the point evaluation

```
Input : Input point $(X : Z)$, degree $\ell$, kernel points $(X_i : Z_i)$ for
        $1 \le i \le (\ell - 1)/2$, bitmask $b \in \{0, 1\}$.
Output: Output point $(X' : Z')$, error variable $error$.
1  $t_+ \leftarrow$ cadd$(X, Z, b)$.
2  $t_- \leftarrow$ csub$(X, Z, b)$.
3  Set $\pi_X \leftarrow 1$, $\pi_Z \leftarrow 1$.
4  for $i \in \{1, \ldots, (\ell - 1)/2\}$ do
5      $t_0 \leftarrow$ cadd$(X_i, Z_i, b)$.
6      $t_1 \leftarrow$ csub$(X_i, Z_i, b)$.
7      $t_0 \leftarrow t_- \cdot t_0$.
8      $t_1 \leftarrow t_+ \cdot t_1$.
9      $t_2 \leftarrow$ cadd$(t_1, t_0, b)$.
10     $t_3 \leftarrow$ csub$(t_0, t_1, b)$.
11     $\pi_X \leftarrow \pi_X \cdot t_2$.
12     $\pi_Z \leftarrow \pi_X \cdot t_3$.
13 $X' \leftarrow$ cadd$(\neg b, X, b)$.
14 $Z' \leftarrow$ cadd$(\neg b, Z, b)$.
15 $X' \leftarrow X' \cdot \pi_X^2$.
16 $Z' \leftarrow Z' \cdot \pi_Z^2$.
17 $error \leftarrow$ cverify$(X', Z', \neg b)$.
18 return $X', Z', error$.
```

unchanged. This method is detailed in Algorithm 15. Note that we are required to run this algorithm twice per isogeny, since both points P_0 and P_1 must be pushed through an isogeny at each step.

In addition to the faults aiming at dummy computations, we need to be able to detect faults in non-dummy computations as well, in order to output an error instead of the output at the end of the algorithm. Otherwise, the attacker could still observe the difference between these cases.

To this end, we note that the output of an isogeny consists of the codomain curve parameters, and the evaluated points. If a fault is injected during the computation of the codomain curve, then (with very high probability) the resulting parameters will not refer to a supersingular curve anymore. This can be deduced from the fact that the probability of a random parameter $a = A/C$ to define a supersingular curve is roughly $1/\sqrt{p}$, and therefore negligible [49]. Thus, the resulting curve at the end of the algorithm will most likely not be supersingular. It therefore suffices to perform a single supersingularity check, e.g. as done in the public key validation in [49], at the end of the algorithm, and output an error in case of a nonsupersingular curve. Instead of using the validation from [49], we use a different, slightly relaxed, approach. We simply sample a random point Q on the curve, and check that $[p+1]Q = \infty$. This method is much faster, but has a small chance to output false positives, so is not usable as public key validation. However, we heuristically tested the probability for false positives, and found that in 10^8 experiments with random curve parameters, our method and the rigorous verifications always had the same result. Thus, it seems to be infeasible for the attacker to exploit this relaxed supersingularity check.

The case of output points will be handled in detail in the following section.

5.1.6.4 Point orders and scalar multiplications

Scalar multiplications take place in line 6 and 9–10 in Algorithm 13, and are intended to produce points of the desired orders. If during such a multiplication a fault is injected randomly, i.e. not aiming to produce a specific faulty output, then the probability for still generating a point of desired order is negligible.[20] The same is true for faults injected during the point evaluation of a real isogeny. For detecting such a fault, it therefore suffices to check if the output point P has the required order ℓ by verifying that $[\ell]P = \infty$.

However, it is not required to perform such a check after each scalar multiplication resp. isogeny. Indeed, it suffices to check point orders in the two following situations:

- At the end of each run through a batch of isogenies, if all computations are running correctly, then both points P_0 and P_1 must be the point at infinity ∞ at the end of the for-loop in line 7 of Algorithm 13. Thus, if we verify this at the end of each run through a batch, we are able to detect faults even if the respective faulty point P_s is not used to generate a kernel input point for an isogeny anymore after the fault is injected. This ensures the correctness of the scalar multiplications in lines 6 and 10 of Algorithm 13, and of the involved isogeny point evaluations.
- In order to validate the scalar multiplication in line 9 of Algorithm 13, we need to verify that K indeed has the correct order ℓ for each isogeny. This is done by calculating $[\ell]K$ and verifying that the result equals ∞ for each isogeny. Note that in the case of real isogenies, a faulty point K leads to wrong results, that can be detected anyway; however, in the dummy case, the input point K is discarded, so this order check is indeed required. In order to keep our algorithm constant-time, this therefore has to be done in both cases.

[20]This follows since the required orders are always small, and for each prime factor $\ell_i | \#E[\pi - 1] = \#E[\pi + 1] = p + 1$, the probability for the order of a random point with x-coordinate in $\mathbb{F}$ to contain the factor ℓ_i is $1 - 1/\ell_i$.

All other scalar multiplications do not require separate order checks, since faults would be detected by the mentioned verifications.

Remark 2 Theoretically, the attacker could try to inject a fault such that a specific output point is produced, although this is not possible in our attacker model. In particular, the above verification does not detect a fault, if the order of the output point divides the desired order. If the attacker produces a point that lies on the same curve as the correct output point would (i.e., not on its twist), then this does not lead to a wrong computation, and therefore does not lead to possible leakage. Note that in CSIDH any point P of order ℓ on the same curve produces the same ℓ-isogeny codomain curve. It only makes a difference if $P \in E[\pi - 1]$ or $P \in E[\pi + 1]$.

This also explains a possible attack strategy: The adversary forces the output of a point on the twist with order dividing the expected order. Thus, the respective isogenies are computed with the wrong direction in the isogeny graph, which leads to leakage.

However, this is only a theoretical attack. Indeed, the chance for this to happen by accident is negligible, and to specifically map a point to a point on the twist of the same order, with an unknown curve, seems infeasible. Even computing such a point with known curve would require to compute an irrational endomorphism, which, if possible in general, would completely break CSIDH on its own [50]. Computing small prime order points, thus possibly having an order dividing the expected order, could otherwise be done through division polynomials. However, as explained above, the attacker does not know the current curve (except for the starting curve) when injecting a fault, which means that division polynomials cannot be computed.

However, there is a simple, but rather costly, countermeasure to prevent attacks of this fashion. It suffices to check if the input kernel generator K lies on the correct curve via a Legendre symbol computation for each isogeny, and output an error otherwise. Although this seems not to be necessary for the reasons above, we report on the performance implications for this in Section 5.1.7.

5.1.6.5 Other functions

The CSIDH constant-time algorithms from [136, 154] feature some more functions outside of the scope of the sections above. Compared to isogeny computations and scalar multiplications, their share of the total running time is small. Nevertheless, we review if countermeasures against fault injections for these functions are required.

The mentioned functions include the Elligator map [21], a method for efficient point sampling. For our discussion and in our implementation, we use the projective Elligator implementation from [51]. Further, the constant-time conditional point swap function `cswap` plays an important role in all current constant-time CSIDH implementations, and we review a method to prevent obvious loop-abort faults.

Apart from these functions, there are more functions, like integer multiplications. However, we disregard them here, since any fault to these functions is detectable through our described methods, e.g. through point order checks.

5.1.6.6 Conditional point swaps

The `cswap` function takes two $\mathbb{F}$-values and a decision bit $b \in \{0, 1\}$ as input, and swaps the input values if $b = 1$. However, this is performed in constant-time, independent from the value of b. The swapping of two elliptic curve points therefore requires two separate executions of `cswap` for their two coordinates.

If one of these swaps is skipped or subject to a fault injection, this means that the respective X- and Z-coordinates of the two points no longer fit together as before. Thus, the coordinates refer to different points then, also leading to different point orders. This also means that our point order checks from the previous sections can detect such a fault.

5.1.6.7 Elligator

The Elligator map as used in [51] efficiently samples projective points $P_0 \in E_A[\pi - 1]$ and $P_1 \in E_A[\pi + 1]$ on the current curve, where the cost is dominated by one Legendre symbol calculation. However, if a fault is injected there, we can no longer guarantee that indeed $P_0 \in E_A[\pi - 1]$ and $P_1 \in E_A[\pi + 1]$. Deviating from this would mean that we compute isogenies with a wrong sign, and therefore obtain a wrong output curve, which can cause leakage.

We mitigate this by computing the Legendre symbol for both of these points, and thereby making sure that $P_0 \in E_A[\pi - 1]$ and $P_1 \in E_A[\pi + 1]$ is satisfied.

5.1.6.8 Loop-abort faults

As mentioned in [51], and also applied to SIDH in [94], loop-abort faults can lead to a stopping of the algorithm, although not all required isogenies have been computed. In the CSIDH implementation featuring dummy isogenies, this can lead to leakage, since a correctly established shared secret in this case means that all the skipped isogenies have been dummies. As usual, this can be prevented by using multiple counters, in order to make it far harder for the attacker to achieve an undetected loop-abort. In the CSIDH implementations from MCR and OAYT, there already are several counters, so it suffices to compare them before outputting the resulting curve, and thereby checking if one of them has been manipulated to abort the loop.

Table 5.2: Performance results for one group action for the CSIDH512 implementation on the ARM Cortex-M4 without and with countermeasures. Averaged over 10 evaluations. Countermeasures for the theoretical twist attack are evaluated separately.

	STM32F407 (24 MHz) [clock cycles]		STM32F303 (7.4 MHz) [clock cycles]	
w/o CM	15 523M		15 721M	
w/ CM (w/o twist)	16 322M		16 751M	
overhead	804M	+5%	1 030M	+7%
w/ CM (w/ twist)	20 907M		21 486M	
overhead	5 384M	+35%	5 765M	+37%

5.1.6.9 Decision bits

In many cases, decision bits must be set, such as b, which decides whether a real or dummy isogeny must be computed, or a decision bit that decides if P_0 or P_1 is used to compute the kernel generator for an isogeny. For our attack models, we could disregard these parts because of the low computational cost, but anyway we provide a simple countermeasure for leakage through an injected fault here. Since in our model the attacker only performs one fault injection, we can simply compute the respective bit twice, check if both computations obtained the same result, and output an error otherwise.

Remark 3 We note that also the dummy-free implementation of [51] offers attack surface; e.g. it is vulnerable to attacks aiming at the `cswap` function, Elligator, or some of the decision bit choices, which means that our discussion on these functions also applies to the dummy-free implementation.

5.1.7 Performance results

We implemented the countermeasures described in Section 5.1.6 into the implementation that was used in Section 5.1.5 to investigate the performance overhead of the proposed countermeasures. The concrete security of CSIDH512 is currently under heavy debate [158, 35]; like most previous work on CSIDH implementations, we focus on the CSIDH512 parameter set. The proposed attacks and countermeasures, however, apply to other parameter sets as well. The code was compiled with `arm-none-eabi-gcc`[21] Version 10.1.0. Table 5.2 contains the performance results without and with the countermeasures implemented. We report cycle counts for both the STM32F303, which is the core on the 32-bit ChipWhisperer Lite, and the

[21]`https://developer.arm.com/`

STM32F407 which is used in various post-quantum cryptography implementations in the literature and the benchmarking project PQM4 [114]. Our benchmarking code is primarily based on PQM4 and we follow the common practice of down-clocking the STM32F407 to 24 MHz to avoid flash wait states impacting the performance results. We report the average over 10 evaluations of the group action. The overhead of the presented countermeasures is 5% to 7% and, therefore, relatively small compared to generic countermeasures like duplicating isogeny computations. The cost for the described twist attack countermeasures is slightly larger, namely 35% to 37% in total, including all other countermeasures. However, as described above, this attack is only of theoretical nature, which means that the former implementation suffices in practice. Note that the implementation of the arithmetic is a portable C implementation that was not heavily optimized for performance for this platform yet. It is, therefore, expected that all implementations can be further improved in terms of speed.

5.2 Safe-Error Attacks on SIKE and CSIDH

This chapter is for all practical purposes identical to the paper *Safe-Error Attacks on SIKE and CSIDH* [42] authored jointly with Juliane Krämer and Marcel Müller, which was published at SPACE 2021.

5.2.1 Introduction

The youngest field of post-quantum cryptography that is studied within NIST's standardization process is isogeny-based cryptography, which was first described in 2006 [66, 164]. Some years later, in 2011, De Feo, Jao, and Plût presented a fast cryptographic scheme based on isogenies, named SIDH (Supersingular Isogeny Diffie-Hellman) [109]. SIDH was used to create the key encapsulation mechanism SIKE (Supersingular Isogeny Key Encapsulation) [108], which was submitted to NIST's standardization process and selected as round 3 alternate candidate, i.e., SIKE is considered promising, but needs to be further studied before being considered for standardization. In 2018, Castryck, Lange, Martindale, Panny, and Renes presented another isogeny-based system, called CSIDH (Commutative Supersingular Isogeny Diffie–Hellman) [49]. Unlike SIKE, CSIDH is non-interactive, making it a potential drop-in replacement for current Diffie-Hellman schemes. CSIDH has not been submitted to NIST's standardization process because it was designed only after the submission deadline had passed. Although the actual security of the suggested CSIDH parameters against quantum attacks was recently questioned [158, 35], CSIDH is still a promising and widely discussed isogeny-based scheme. However, the recent quantum attacks show that the young field of isogeny-based cryptography has not been sufficiently studied with respect to (quantum) cryptanalysis yet. Also, the physical security of isogeny-based schemes has not been sufficiently studied yet.

In this work, we analyze the physical security of SIKE and CSIDH. Physical attacks allow attackers to deduce secret information of an algorithm by observing or modifying the platform it operates on. In a passive (or side-channel) attack, the attacker analyzes physical information that they can measure while cryptographic operations are computed. In an active attack, on the other hand, the attacker directly interacts with the running algorithm, causing a change in its operations through which information can be extracted. Hence, active attacks are also called fault attacks.

Analyzing SIKE and CSIDH with respect to a specific fault attack is the focus of this work. We analyze both schemes regarding their vulnerability towards *safe-error attacks*. Safe-error attacks have been first published by Yen and Joye in 2000 [188]. They suggested that by inducing transient faults, an implementation leaks one bit of information depending on whether the algorithm results in an error or not. Yen and Joye first described attacks on smart cards using a square-and-multiply algorithm and later applied safe-error attacks on the Montgomery ladder, showing that by perturbing memory during computation, one can deduce one bit of secret information [112]. Safe-error attacks are particularly interesting because even if the algorithm were to detect a fault in its operation, it will still leak information. Hence, standard countermeasures, like checking for faults and outputting a random value in case a fault was detected, still provide the attacker with information and therefore are not sufficient to protect an implementation against safe-error attacks.

Safe-error attack mitigations usually do not feature in current implementations, rendering them vulnerable against these attacks, see, e.g., [26]. Also our work shows that recent implementations of isogeny-based schemes do not provide explicit protection against safe-error attacks. This is concerning especially since some of our attacks are similar to attacks that have long been known in the ECC community, e.g., [188, 112].

Our Contribution. The focus of this work is to analyze SIKE and CSIDH with respect to safe-error attacks. To the best of our knowledge, SIKE has not been studied with respect to these attacks before.

We develop attack scenarios for SIKE and CSIDH and demonstrate the feasibility of the presented safe-error attacks by performing practical experiments. The experiments were performed against C implementations of SIKE and CSIDH on a ChipWhisperer board with an ARM Cortex-M4 processor as target core. The implementation of CSIDH that we attacked is a constant-time implementation based on dummy isogenies. We achieve full key recovery of all n bits of the secret key within $O(n)$ interactions for two of the four attacks laid out in this paper. We discuss possible countermeasures and their performance impact. The code used for this work is available[22] in the public domain, which includes the modified CSIDH and SIKE Cortex-M4 implementation and all attack scripts.

The attack against SIKE that we carried out practically can analogously be applied to B-SIDH [60].

Related work. Although isogeny-based cryptography provides promising candidates for quantum-resistant public-key schemes, only few results regarding the physical security of isogeny-based cryptography in general and SIDH [120, 91, 94, 180], SIKE [193], and CSIDH [40, 51, 123] in particular exist. In [91], the authors presented the first fault attack on SIDH, together with corresponding countermeasure. In [120], the authors propose different zero-value attacks on SIDH. Based on loop-abort fault injection, Gélin and Wesolowski presented side-channel and fault attacks against isogeny-based primitives [94]. The first published physical attack on SIKE was a power side-channel attack exploiting differences in calculations depending on the secret key [193]. Ti proposed in [180] a fault attack on SIDH by changing the

[22]`https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC`

base point to a random point via fault injection. In [177], the authors presented the first experimental realization of Ti's theoretical fault attack and proposed countermeasures against this attack. In [51], the authors analyzed CSIDH for potential attacks by reviewing and improving the constant-time implementations of [136] and [154]. Furthermore, they proposed a dummy-free CSIDH algorithm. A recent work [40] presents safe-error and further fault attacks, together with countermeasures, on a constant-time CSIDH implementation with dummy isogenies. The attack against CSIDH that we carried out practically attacks the resulting implementation of [40]. LeGrow and Hutchinson [123] suggest to randomize the order of execution of isogenies to increase the number of attacks required when attacking dummy-based constant-time implementations of CSIDH.

Concurrently to our work, several PQC schemes have been analyzed with respect to safe-error attacks [26]. However, isogeny-based schemes are not covered in this work.

Organization. In Section 5.2.2, we present necessary background on SIKE, CSIDH, and safe-error attacks. In Sections 5.2.3 and 5.2.4, we present safe-error attacks on SIKE and CSIDH, respectively. In Section 5.2.5, we explain how to perform the described safe-error attacks on a real device and present full key recovery. We discuss possible countermeasures in Section 5.2.6 and conclude this work in Section 5.2.7.

5.2.2 Background

We first discuss implementation details of SIKE and CSIDH. For readers not familiar with isogenies, we refer to [59]. Afterwards, the introduction to safe-errors shows the pattern common to the attacks and how they work.

5.2.2.1 SIKE

SIKE (Supersingular Isogeny Key Encapsulation) is an interactive key encapsulation using supersingular elliptic curves [108]. SIKE has passed into the third round of the NIST process[23] as alternate candidate for future standardization. To achieve the goal of becoming standardized it will need to be studied further, especially with respect to efficiency improvements and all aspects of misuse resistance. SIKE uses SIDH internally, and SIDH will be the main target of the attacks presented in the following section. For a detailed overview of SIDH as used in SIKE, we refer to [108].

SIDH is constructed as follows: A public prime $p = 2^{e_2}3^{e_3} - 1$ such that $2^{e_2} \approx 3^{e_3}$ is chosen, as well as two points on the torsion group associated to their base: $P, Q \in E_0[2^{e_2}]$ or $E_0[3^{e_3}]$. These represent the respective public generators. The rest of the algorithm is computed over $\mathbb{F}_{p^2}$. At the start of

[23]`https://doi.org/10.6028/NIST.IR.8309`

```
// Main loop
for (i = 0; i < nbits; i++) {
    bit = (m[i >> LOG2RADIX] >> (i & (RADIX-1))) & 1;
    swap = bit ^ prevbit;
    prevbit = bit;
    mask = 0 - (digit_t)swap;

    swap_points(R, R2, mask);
    xDBLADD(R0, R2, R->X, A24);
    fp2mul_mont(R2->X, R->Z, R2->X);
}
```

Listing 5.1: LADDER3PT – SIKE

the exchange, each party agrees on picking a base of either 2 or 3 as long as they differ between them. Afterwards, each party generates a private key $\mathsf{sk} \in \mathbb{F}_{p^2}$. Of note here is that in the efficient implementation of [108], three points are used. The third point is $R = P - Q$ and is used to speedup the computation through a three-point ladder [83](cf. Algorithms 16 and 19, and Listing 1). Using these generators as well as their private key, each party then computes their public curve E_2 or E_3. This curve is calculated through a chain of e_2 2-isogenies, or e_3 3-isogenies respectively. Each isogeny uses a generator of the form $\langle P + [\mathsf{sk}]Q\rangle$ as the kernel. The projection of the other party basis point and this curve are then sent to the other party, where the same procedure is repeated to arrive at the curve $E_{2/3}$ and $E_{3/2}$. These two curves are isomorphic to each other and thus the parties have arrived at a shared secret: the j-invariant of $E_{2/3}$ and $E_{3/2}$, respectively.

The submitted implementation from Round 3 is constant-time and already includes several countermeasures against fault attacks. The implementation is secure against the attack presented in Section 5.2.3.1, but vulnerable to the second one as presented in Section 5.2.3.2.

5.2.2.2 CSIDH

CSIDH (Commutative Supersingular Isogeny Diffie–Hellman) describes a non-interactive key exchange using supersingular elliptic curves [49]. For a more detailed overview of the key exchange, we refer to [49].

CSIDH is constructed as follows: A prime p is chosen of the form $p = 4 \cdot \ell_1 \cdots \ell_n - 1$, where the ℓ_i are small pairwise distinct odd primes. The rest of the algorithm is computed in $\mathbb{F}_p$. The algorithm uses elliptic curves in Montgomery form: $E_0 : y^2 = x^3 + Ax^2 + x$. To begin, each party generates a secret key $(e_1, \dots, e_n)$, where each e_i is sampled uniformly random from the interval $[-m, m]$ with $m \in \mathbb{N}$. The key exchange is then prepared by calculating the elliptic curve associated with the secret key: For each e_i

Algorithm 16: xDBLADD

```
1  function xDBLADD
       Input: $(X_P : Z_P), (X_Q, Z_Q), (X_{Q-P} : Z_{Q-P})$, and
              $(a^+_{24} : 1)$ $(A + 2C : 4C)$
       Output: $(X_{[2]P} : Z_{[2]P}), (X_{P+Q}, Z_{P+Q})$
2      $t_0 \leftarrow X_P + Z_P$
3      $t_1 \leftarrow X_P - Z_P$
4      $X_{[2]P} \leftarrow t_0^2$
5      $t_2 \leftarrow X_Q - Z_Q$
6      $x_{P+Q} \leftarrow X_Q + Z_Q$
7      $Z_{[2]P} \leftarrow t_1^2$
8      $t_1 \leftarrow t_1 \cdot X_{P+Q}$
9      $t_2 \leftarrow X_{[2]P} \cdot -Z_{[2]P}$
10     $X_{[2]P} \leftarrow X_{[2]P} \cdot Z_{[2]P}$
11     $X_{P+Q} \leftarrow a^+_{24} \cdot t_2$
12     $Z_{P+Q} \leftarrow t_0 - t_1$
13     $Z_{[2]P} \leftarrow X_{P+Q} + Z_{[2]P}$
14     $X_{P+Q} \leftarrow t_0 + t_1$
15     $Z_{[2]P} \leftarrow Z_{[2]P} \cdot t_2$
16     $Z_{P+Q} \leftarrow Z^2_{P+Q}$
17     $X_{P+Q} \leftarrow X^2_{P+Q}$
18     $Z_{P+Q} \leftarrow X_{Q-P} \cdot Z_{P+Q}$
19     $X_{P+Q} \leftarrow Z_{Q-P} \cdot X_{P+Q}$
20     return $(X_{[2]P} : Z_{[2]P}), (X_{P+Q}, Z_{P+Q})$
```

Algorithm 17: CSIDH Algorithm of [154]

```
Input: $A \in \mathbb{F}_p, m \in \mathbb{N}$, a list of integers $(e_1, \dots, e_n) \in [-m, m]^n$ and $n$
       distinct odd primes $\ell_1, \dots, \ell_n$ s.t. $p = 4\prod_i \ell_i - 1$.
Output: $B \in \mathbb{F}_p, m \in \mathbb{N}$ s.t. $E_B = (\mathfrak{l}_1^{e_1} \cdots \mathfrak{l}_n^{e_2}) * E_A$, where
        $\mathfrak{l}_i = (\ell_i, \pi - 1)$ for $i = 1, \dots, n$, and $\pi$ is the $p$-th power
        Frobenius endomorphism of $E_A$.
1  Set $e_i' = m - |e_i|$ for $i = 1, \dots, n$
2  while some $e_i \neq 0$ or $e_i' \neq 0$ do
3      Set $S = \{i | e_i \neq 0$ or $e_i' \neq 0\}$
4      Set $k = \prod_{i \in S} \ell_i$
5      Generate points $P_0 \in E_A[\pi + 1]$ and $P_1 \in E_A[\pi - 1]$ by Elligator
6      Let $P_0 \leftarrow [(p+1)/k]P_0$ and $P_1 \leftarrow [(p+1)/k]P_1$
7      for $i \in S$ do
8          Set $s$ the sign bit of $e_i$
9          Set $Q = [k/\ell_i]P_s$
10         Let $P_{1-s} \leftarrow [\ell_i]P_{1-s}$.
11         if $Q \neq \infty$ then
12             if $e_i \neq 0$ then
13                 Compute an isogeny $\phi : E_A \to E_B$ with $\ker \phi = \langle Q \rangle$
14                 Let $A \leftarrow B, P_0 \leftarrow \phi(P_0), P_1 \leftarrow \phi(P_1)$, and
                     $e_i \leftarrow e_i - 1 + 2s$
15             else
16                 Dummy computation
17                 Let $A \leftarrow A, P_s \leftarrow [\ell_i]P_s$, and $e_i' \leftarrow e_i' - 1$.
18         Let $k \leftarrow k/\ell_i$
19 return $A$
```

a total of $|e_i|$ ℓ_i-isogenies have to be calculated. The sign of e_i represents the direction taken in the respective ℓ_i-isogeny graph. As the composition of isogenies is commutative, each computed curve will be isomorphic no matter in which order they are calculated. The isognies are then chained to compute the public curve associated to the secret key: $E_0 \xrightarrow{(e_1,\dots,e_n)} E_A$. Bob does the same to calculate E_B. The parameter of the curves E_A and E_B correspond to the public keys and are then exchanged and each party repeats their isogeny calculation using the other's public key as the starting curve: Alice calculates $E_B \xrightarrow{(e_1,\dots,e_n)} E_{BA}$ and Bob calculates E_{AB} in a similar fashion. The final curves E_{BA} and E_{AB} are the same, and the shared secret is the A parameter of this curve in Montgomery form.

The straightforward implementation of the algorithm would be highly variable in time, since different amounts of isogenies need to computed, depending on the secret key. It would be easy for an attacker to trace the amount of isogenies calculated and their degree as isogenies with a larger degree require more computational effort. In 2019, Meyer, Campos, and Reith have presented a constant-time implementation of CSIDH [136]. The authors tackle this issue by making the amount of isogeny evaluations constant, thus only leaking the degree of the isogenies themselves and not the exact number of them. This follows from the aforementioned fact that higher degree isogenies take longer to construct and, e.g., could be recovered through a timing attack. They achieve this by calculating "dummy" isogenies which serve as extra computational time to thwart timing attacks from finding the real amount of isogenies of a given degree. Further, they change the interval from which the secret key parts are sampled from $[-m, m]$ to $[0, 2m]$ so that an attacker cannot tell apart secret keys with unbalanced positive and negative parts. Unfortunately, these dummy calculations have added a new attack vector: loop-abort attacks. Such an attack was first described in passing in [51]. In [40] the approach using dummy isogenies has been further refined. Thereby, the authors analyzed the constant-time implementation for fault-injection attacks. This resulted, among others, in added safeguards to the point evaluation and codomain curve algorithm. However, these safeguards do not protect against the attack described in Section 5.2.4.1, as the attacker assumed in this paper has a different threat model.

Following [136] in [154] the authors proposed to speed-up the implementation by reverting the secret key part interval to $[-m, m]$ and guarding against unbalanced keys by using two points instead of one [154]. This change, however, has introduced a possible new attack vector as described in Section 5.2.4.2.

5.2.2.3 Safe-Error Attacks

In [188], Yen and Joye introduce a new category of active attacks, so called safe-error attacks. In this kind of attacks, the adversary uses fault injections to perturb a specific memory location with the intent of not modifying the final result of the computation: the algorithm may overwrite or throw away modified values, making them "safe errors". The presence or absence of an error then gives insight into which codepath the algorithm executed. Two kinds of safe-error attacks exist: in a memory safe-error (M safe-error) attack, the attacker modifies the memory, i.e., in general these attacks focus on specific implementations [112, 189, 190]. In a computational safe-error (C safe-error) attack, however, the computation itself is attacked through, e.g., skipping instructions. Hence, C safe-error attacks rather target algorithmic vulnerabilities [112, 190].

The general construction of a safe-error attack is as follows:

Algorithm 18: A toy algorithm vulnerable to a variable-access attack

```
Input: $S$ the $n$-bit secret key
Output: a public message $M$
1  $M \leftarrow 1$
2  $K \leftarrow 0$
3  $P \leftarrow 0$
4  for $i \in 0..n$ do
5      if $S_i = 0$ then
6          $K \leftarrow \text{calculate}(S_i, P, K)$
7      else
8          $P \leftarrow \text{calculate}(S_i, K, P)$
9      $M \leftarrow M + K * P$
10 return $M$
```

Suppose an algorithm iterates over secret data. It then branches and does slightly different calculations depending on whether a given bit in the secret data is equal to 0 or 1. The algorithm presented in Algorithm 18 has been secured against timing side-channel attacks by consuming the same time in each branch. Precisely this predictability, enforced to thwart timing attacks, makes safe-error attacks easier to carry out, as these attacks require timed fault-injections. When implementing countermeasures, implementers thus have to investigate all implications that these countermeasures have. However, since some side-channel attacks, e.g., timing attacks, are generally easier to carry out than other physical attacks, e.g., safe-error attacks, it can still be the right decision to fix a specific vulnerability by enabling other, practically less relevant attacks. In application-related implementations, explicit branching on secret data is usually avoided. However, the different memory access patterns still occur due to the structure of the respective algorithm. As we show in Section 5.2.3.1, using a constant time swap algorithm instead of condition branching is not sufficient and may even provide an additional attack vector.

Analyzing the read and write patterns of Algorithm 18 and classing them according to the state that they occur in allows to look for differences that could be exploitable. These differences can be rendered in a table, such as Table 5.3. This allows for visual inspection of differences.

This representation makes it immediately clear that even though the same method is being called, it affects different data. This allows an attacker to exploit the difference between the two branches by modifying one memory location and checking whether a safe-error occurred.

Table 5.3: Access patterns depending on the i-th bit of the secret key

Condition	Read Variables	Written Variables
$S_i = 0$	P, K	K
$S_i = 1$	P, K	P

Example: Let's assume we try to attack the first branch, when $S_i = 0$. During the calculate routine, we modify the memory used by the variable K in such a way that it does not change the result of the computation. This is done by perturbing the memory once the given memory location is not read anymore, but before it is being potentially written to. After the calculate routine has executed, either K or P has been overwritten. If our guess of $S_i = 0$ was correct, due to being overwritten after being perturbed by the fault, K now holds again correct information in context of the algorithm. Letting the algorithm finish leaks the information whether our guess was correct: If it finishes normally, S_i was indeed 0. If we assume that M is known and verifiable, we can check to see if the outcome was wrong, or, simpler, an error occurred. If either happened, then S_i was 1, as the faulted K did not get overwritten and subsequently changed the calculation. This attack needs to be then repeated n times to fully recover the secret key S.

5.2.3 Attacks on SIKE

In this section, we analyze the implementation of SIKE submitted to round 3 of NIST's standardization process [108] in the context of safe-error attacks. First, we describe a memory safe-error attack in Section 5.2.3.1, then we describe a computational safe-error attack in Section 5.2.3.2. For both attacks, we assume that the victim has a static secret key. Both the encapsulator and the decapsulator can be the victim of this attack.

5.2.3.1 M-Safe Attack on SIKE

We first give a high-level overview on how the attack is constructed. Then, we give a more detailed analysis of the individual steps of the attack.

As shown in Section 5.2.2.1 each SIKE participant has their own secret key $m \in \mathbb{F}_{p^2}$. This key is used to calculate the subgroup $\langle P, [m]Q \rangle$ representing the kernel of their secret isogeny. The point multiplication $[m]Q$ is performed through a three-point ladder algorithm as seen in Algorithm 19. Important here is that the **LADDER3PT** function is called with the secret key m as the first argument. The attacker requires the following capabilities: They need to be able to introduce a memory fault during a specific point of execution, as well as be able to verify the result of a given SIKE run. Both the shared secret as well as any execution errors need to be known afterwards. The attack proposed in this section then follows three parts:

Algorithm 19: The 3-Point Ladder

```
1 function LADDER3PT
      Input: $m = (m_{l-1}, ..., m_0)_2 \in \mathbb{Z}, (x_P, x_Q, x_{Q-P})$, and $(A : 1)$
      Output: $(X_{P+[m]Q} : Z_{P+[m]Q})$
2     $((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2)) \leftarrow ((x_Q : 1), (x_P : 1), (x_{Q-P} : 1))$
3     $a_{24}^{+} \leftarrow (A + 2)/4$
4     for $i = 0$ to $l - 1$ do
5         if $m_i = 1$ then
6             $((X_0 : Z_0), (X_1 : Z_1)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_1 : Z_1), (X_2 : Z_2), (a_{24}^{+} : 1))$
7         else
8             $((X_0 : Z_0), (X_2 : Z_2)) \leftarrow$ xDBLADD$((X_0 : Z_0), (X_2 : Z_2), (X_1 : Z_1), (a_{24}^{+} : 1))$
9     return $(X_1, Z_1)$
```

Table 5.4: Access patterns depending on the ith-bit of the secret key

Condition	Read Variables	Written Variables
$m[i] = 0$	$(X_0, Z_0), (X_1, Z_1), (X_2, Z_2)$	$(X_0, Z_0), (X_2, Z_2)$
$m[i] = 1$	$(X_0, Z_0), (X_1, Z_1), (X_2, Z_2)$	$(X_0, Z_0), (X_1, Z_1)$

With the goal of extracting an n-bit secret key, the attacker

1. initiates a SIKE key agreement,
2. introduces a memory fault of any kind (bit-flip, scrambling,...) during the i-th iteration of **LADDER3PT**, and
3. uses the result of the SIKE run to obtain the value of the i-th bit of the secret key.

Steps 1 to 3 have to be repeated n times to reconstruct the complete secret key.

In detail, this means that the attack on this three-point ladder algorithm follows the schema as described in Section 5.2.2.3. Depending on a given bit of the secret key, different variables are modified. This can be seen in Table 5.4. In this case either (X_1, Z_1) or (X_2, Z_2) are passed to **xDBLADD**. Without loss of generality, let's assume for the rest of this section that we attack m and that the guess for the i-th bit is $m[i] = 1$. By following the general outlines of a safe-error attack one needs to modify (X_1, Z_1) between its last use and the moment it gets written to. Such a moment exists in Algorithm 19 Line 6 (cf. Section 5.2.2.1): (X_1, Z_1) is passed to the **xDBLADD** subroutine as the *second* argument, thus $(X_Q, Z_Q) = (X_1, Z_1)$ in Algorithm

16 (cf. Section 5.2.2.1). (X_1, Z_1) is passed as the *third* argument in Line 8, this difference is dependent on the secret key. The **xDBLADD** method (as seen in Line 6 in Algorithm 16) then returns two values, one of which is assigned to (X_1, Z_1) in Algorithm 19. In the **xDBLADD** routine from Line 6 onwards, (X_Q, Z_Q) is no longer read, and thus the value of (X_1, Z_1) stays unused until the function returns. This is where the attacker executes the active attack, by scrambling the values backing (X_Q, Z_Q), i.e., (X_1, Z_1). If the attack on the memory location of (X_1, Z_1) was successful and our guess was correct, the algorithm will, upon return, overwrite our modification and finish without encountering an error. One can thus conclude that $m[i] = 1$. Should our guess of $m[i] = 1$ be incorrect, then the algorithm computes a mismatching shared secret or raises an error. In this case, $m[i] = 0$. Either way, a single bit of information is gained of the secret key. Consequently, all n bits of the static secret key m can be read by this method and the full key can be recovered through n runs of this attack. The complete attack thus consists of these steps:

1. The attacker observes a normal SIKE key agreement.
2. As **xDBLADD** gets called during **LADDER3PT**, overwrite (X_1, Z_1) on the i-th iteration and observe the final result.
3. If the SIKE de/encapsulation fails, we know that (X_1, Z_1) did *not* get overridden. Thus $m[i] = 0$ otherwise $m[i] = 1$.

Repeat steps 1 to 3 n times to recover the complete n-bit secret key.

The SIKE implementation in [108] has several parameter sets, each influencing the range of possible values of the secret key. For example, SIKEp610 has an exponent $e_2 = 305$ with an estimated NIST security level 3 [108]. The private key m is thus sampled from $\{0, ..., 2^{305} - 1\}$, giving the private key 305 bits of total length. Therefore an attacker, trying to attack a SIKEp610 instantiation, would need to repeat the attack at least 305 times to achieve full key recovery.

In the latest version of **xDBLADD**, as published for the third NIST PQC process round [108], the authors have chosen to use a simultaneous double-and-add algorithm. This implementation prevents this particular attack as there is no moment during execution that P or Q is written before it is potientially read. This is also true during compilation: the order of operations in the assembly stays the same. Nonetheless, future implementations have to make sure that they are not vulnerable when using a different algorithm.

5.2.3.2 C-Safe Attack on SIKE

Similar to the M safe-error attack on SIKE described in the previous section, the attack described in this section exploits the difference in memory accesses depending on a bit of the secret key. Again, each party generates their own private key m, used to generate the subgroup $\langle P + [m]Q \rangle$ of their private isogeny (cf. Section 5.2.2.1). This point multiplication $P + [m]Q$ is done through a three-point ladder as seen in Algorithm 19 and Listing 1 (cf. Section 5.2.2.1). In the C implementation, published in [108], the authors use a constant-time swapping algorithm to exchange the points R and $R2$ depending on the i-th bit of the secret key (see Line 359 of Listing 1). The function is called `swap_points` and accepts both points and a mask as input. We denote the i-th bit of the secret key as $m[i]$. The mask of the swapping function is calculated as $\text{xor}(m[i], m[i-1])$, with a starting value of 0 for $m[i-1]$ if $i = 0$. If the mask is 1 the points are exchanged, otherwise they are left as is. This behavior can be exploited by meddling with this function call. It could for example simply be skipped, or the computation of the mask be perturbed such that on a 0 mask it stays 0, but on a 1 mask the value is randomized. Assuming $\text{xor}(m[i], m[i-1])$ and an attacker skips this function call using an active attack on the i-th loop, the end result will be unchanged. If the value had been $\text{xor}(m[i], m[i-1])$, then the end result would be wrong, as the wrong point would have been used for the rest of the calculation.[24] Since we know that in the first iteration $m[i-1]$ is forced to 0, the mask is simply set to the value of $\text{xor}(m[0], 0) = m[0]$. The second iteration of attack then knows the value of $m[0]$ and so on. Thus, in general the bit $m[i]$ is leaked through a C safe-error. As the keyspace for m is equal to $[0, ..., 2^{e_2} - 1]$, similar to the attack in Section 5.2.3.1, the attack needs to be repeated at least 305 times to achieve full key recovery when the parameter set SIKEp610 is used.

5.2.4 Attacks on CSIDH

In this section, we analyse CSIDH with respect to safe-error attacks. We analyse two recent implementations of CSIDH [40, 154]. Both implementations are constant-time implementations, and both implementations achieve this kind of timing attack resistance through dummy isogeny computations. The main difference between both implementations is that in [40], computations are done on one point only, while in [154], two points are used. The analysis of both implementations with respect to safe-error attacks is presented in Section 5.2.4.1 and Section 5.2.4.2, respectively. For both of the presented attacks, it is assumed that the victim uses a static secret key.

5.2.4.1 M Safe-Error Attack on an Implementation Using One Point

In [40], the authors have evaluated possible physical attack vectors for CSIDH implementations using dummy isogenies. One threat model they

[24]Wrong shared secret or an error raised from the algorithm.

Table 5.5: Access pattern depending on the secret key e during the key exchange

Condition	Read Variables	Written Variables
$e_i \neq 0$	P_0, P_1	P_0, P_1
$e_i = 0$	P_s where s is the sign bit of e_i	P_s

did not consider, is one that can introduce memory faults. This will be the focus of the attack in this section. The attacker only needs to be able to change a single bit in a certain byte range. In [40], during the execution of a dummy isogeny, the curve parameter A is not modified. If however a non-dummy isogeny is calculated, then the A parameter is changed corresponding to the newly calculated curve. This leads to a possible attack vector: assume without loss of generality that the algorithm is currently calculating isogenies of degree ℓ_i. If it is currently calculating a dummy isogeny, a new parameter A is computed, but *directly discarded*. If a real isogeny is calculated, that result is then used further. A fault injected with the intent of modifying the parameter A can now discern if a real or dummy isogeny is being calculated: if one attacks a real isogeny, the modified value will be propagated and cause a mismatch of the final shared secret. If it was a dummy isogeny however, the modified A was discarded and the shared secret is not impacted. This is now repeated for each possible value of e_i, so as to find out the first time a dummy isogeny is calculated. The value of e_i is then the amount of real isogenies that have been calculated for ℓ_i. In the implementation in [40] e_i is sampled from the range $[0, 10]$, therefore one needs on average 5 attacks per e_i to recover its value. In CSIDH-512 of [40] the secret key has 74 components, thus on average, an attacker would need to run $5 \times 74 = 370$ attacks to recover the full key.

5.2.4.2 M Safe-Error Attacks on an Implementation Using Two Points

In [154], the authors have introduced a new algorithm that uses two points to calculate the CSIDH action. This version has an issue similar to the one described in Section 5.2.4.1, where the parameter A is discarded when calculating a dummy isogeny. Thus it has also the potential for an M safe-error attack by attacking the A parameter assignment. Unlike the implementation in [40], in [154] the range $[-5, 5]$ is used for each e_i. Even though an attacker additionally needs to recover the sign of e_i now, this reduces the amount of overall attacks required to recover a single e_i.

Further, the CSIDH action as described in [154] has another M safe-error attack vector that will be explained in this section. Table 5.5 shows the access patterns of two different variables depending on a part of the secret key: only one point is overwritten when e_i equals 0 during the CSIDH action calculation at Line 17 in Algorithm 17 (cf. Section 5.2.2.2). This opens up the potential of perturbing a given P_0 or P_1 and finding out if this had any effect on the calculation. If there was no effect, then the sign of e_i is equal to the index of the point that was overwritten: 0 if positive, 1 if negative. This allows the attacker to find the sign of a specific e_i since the dependency between isogenies of degree ℓ_i and its running allows for attacking a specific degree ℓ_i [49]. Now let s_i be the sign of e_i. In total, Algorithm 17 does e_i calculations of isogenies of order ℓ_i. After each calculation, it decrements e_i to keep track of how many more real isogenies need to be computed. Once $e_i = 0$, only dummy operations are executed. The task is thus, to find out how many real isogenies are calculated. One can run the following procedure to find the value of e_i: Start with $n = 0$. Modify P_{s_i} after n iterations just before it is potentially overwritten, and check the final result. If the shared secret is correct or n is larger than the maximal possible value for e_i, we know $e_i < n$ at that point and we can stop the process, otherwise $e_i > n$, increment n and retry. Once this procedure terminates, e_i equals the amount of calculated real isogenies. Applying this procedure repeatedly, one can deduce the whole secret key $(e_1, \ldots, e_n)$. As [154] uses an instantiation where the private key elements can range from -5 to 5, in total $2.5 + 1 = 3.5$ attacks are required per e_i, as well as finding s_i. In that instantiation, 74 elements are used per secret key, therefore an attacker would need to run $74 \times 3.5 = 259$ attacks on average for the signs and the full key recovery in total. The attack can be summarised as follows:

1. Reveal which ℓ_i is currently being computed from the length of computation.
2. On Line 17 in Algorithm 17 only P_{s_i} is being assigned. Thus, perturbing the memory of P_{s_i} while $[\ell_i]P_{s_i}$ is being calculated will allow to deduce whether $i = 0$, or $i = 1$. From now on, we assume that s_i is known for each e_i.
3. Knowing the sign allows us to now explicitly attack either P_0 or P_1 and thus find out whether a real or dummy isogeny is being calculated.

If the final shared secret is correct, it was a real isogeny, otherwise it was a dummy. The value of e_i is equal to the count of real isogenies. Once all e_i and their signs s_i have been recovered, the full private key $(e_1, \ldots, e_n)$ can be reconstructed.

5.2.5 Practical Experiments

In this section, we explain how to perform the described attacks on a ChipWhisperer board and present the achieved security impact. In the case of SIKE, we present full key recovery. In the case of CSIDH, due to the relatively long runtime on the target architecture ($\approx$ 7 seconds for the reduced version of CSIDH), we calculated the maximum number of possible runs in advance and determined further attack parameters accordingly.

All practical attacks were implemented using the ChipWhisperer tool chain[25] (version 5.3.0) in Python (version 3.8.2) and performed on a ChipWhisperer-Lite board with a 32-bit STM32F303 ARM Cortex-M4 processor as target core. Based on available implementations, we wrote slightly modified ARM implementations of SIKEp434 and CSIDH512 to make them suitable for our setup. Security-critical spots remained unchanged. All binaries were build using the GNU Tools for ARM Embedded Processors 9-2019-q4-major[26] (gcc version 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]) using the flags: `-Os -mthumb -mcpu=cortex-m4 -mfloat-abi=soft`. The code is freely available at `https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC` and `https://doi.org/10.5281/zenodo.6900027`.

In all attack models the adversary aims to attack the calculation of the shared secret in order to learn parts of the private key. The shared secrets are calculated without randomness, i.e., points and private keys used were computed in advance. Both in the case of SIKE and CSIDH, the adversary is able to randomise variables or skip instructions by injecting one fault per run. Furthermore, we assume that the attacker is able to trigger and attack the computation of the shared secret multiple times using the same pre-computed private keys. However, in a real environment the attacker is limited to observe the impact of a fault injection (whether both shared secrets are equal or not), by noticing possible unexpected behaviour in the protocol. Although static keys are mostly used in server environments where such invasive fault attacks are not feasible, in [153], the authors described exemplary environments and challenge-response scenarios where such static-key attacks can be deployed. Furthermore, CSIDH provides a non-interactive (static-static) key exchange with full public-key validation.

5.2.5.1 Attacks on SIKE

Since the current implementation [108] is immune to the attack described in Section 5.2.3.1, we focus on the attack explained in Section 5.2.3.2. As described, the adversary deploys safe-error analysis to recover the private key during the computation of the three-point ladder. Since the attacked algorithm runs in constant time, an attacker can easily locate the critical spot, which in our case represents the main loop within the ladder computation. Thus, an attacker who can accurately induce any kind of computational fault inside that spot at the i-th iteration, may be able to deduce if the i-th bit of the private key is set or not, i.e, $\mathsf{sk}_i = 0$ or $\mathsf{sk}_i = 1$ according to whether the resulting shared secret is incorrect or not. Thus, in this model the required

[25]`https://github.com/newaetech/chipwhisperer`, commit fa00c1f
[26]`https://developer.arm.com/`

number of injections for a full key recovery only depends on the length of the private key. In this setup, the fault is injected by suddenly modifying the clock (clock glitching), thus, forcing the target core to skip an instruction.

The SIKEp434 Cortex-M4 implementation[27] from [167] available at the pqm4 project [114] provided the basis for our implementation. However, this attack can be applied to all available software implementations of SIKE[28] including the round-3 submission [108] to NIST's standardisation process. More precisely, the code part that represents this vulnerability remains the same across all available implementations.

Results. We assume that the attacker knows critical spots within the attacked loop (cf. Listing 1) which reveal one bit of the private key after a single fault injection with high accuracy. As shown in this work, such spots and the corresponding suitable parameters for the injection (e.g., width and internal offset of the clock glitch) can be empirically determined in advance with manageable effort.

In order to determine the success rate for each individual of the 218 bits of the private key, we performed 21,800 fault injections (100 injections for each bit) and achieved a relatively high accuracy. More precisely, we obtained on average over all bits 100% (leading to an error probability $p_0 = 0$, as denoted in Figure 5.4) accuracy for the case $\mathsf{sk}_i = 0$ and an accuracy of over 86% (denoted as p_1 in Figure 5.4) for the case $\mathsf{sk}_i = 1$. As shown in Figure 5.4, only 5 fault injections are required for each bit, thus 1,090 injections in total to achieve a success rate above 99% for full key recovery. Since in our inexpensive setup a single run takes about 12 seconds, full key recovery requires about 4 hours.

5.2.5.2 Attacks on CSIDH

Since the practical implementation is similar for both attacks, we show without loss of generality how we realised the attack described in Section 5.2.4.1. The attacker aims to distinguish a real from a dummy isogeny. For this, they inject a fault during the computation of an isogeny and observe if it impacts the resulting shared secret. In this attacker model the adversary can target isogeny computations at positions of their choice and is further able to trace the faulty isogeny computation to determine its degree. Due to non-constant time computation within the calculation of the isogeny (e.g., a square-and-multiply exponentiation based on the degree [136, 154, 40]), the degree of a given isogeny might be recovered with manageable effort, e.g., using Simple Power Analysis [119].

In our setup, the fault is injected by temporarily under-powering the target core, i.e., by reducing for some clock cycles the value of the supply

[27]`https://github.com/mupq/pqm4`, commit 20bcf68
[28]`https://sike.org/#implementation`

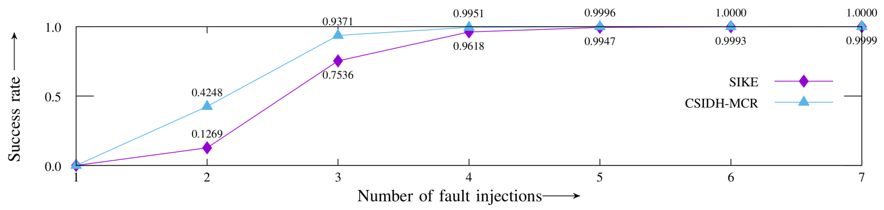


Figure 5.4: Success rate for full key recovery as a function of the number of fault injections per bit (SIKE) or isogeny (CSIDH), respectively. Let α be the number of injections for each bit/isogeny. Since a single faulty shared secret is sufficient to distinguish the cases, the success rate for full key recovery can be calculated by $P(\alpha) = [(0.5 \cdot (1 - B(0, \alpha, p_1))) + (0.5 \cdot B(0, \alpha, p_0)))]^\lambda$, where λ equals the number of bits in the case of SIKE and equals $\sum_{i=1}^{n} \lceil \log_2(m_i) \rceil$ for all m_i of the corresponding bound vector $\mathtt{m} = (m_1, m_2, \ldots, m_n)$ in the case of CSIDH, $B(k, n, p) = \binom{n}{k} \cdot p^k (1-p)^{n-k}$, and p_0, p_1 correspond to the respective probabilities.

voltage of the attacked device below the minimum value the device is specified for. Such an attack might lead to an unpredictable state in the target variable during an assignment and can therefore be applied to attack the vulnerable spot regarding the co-domain curve A, as defined in Section 5.2.4.2. For illustration, the attacks occur during the calculation of the first isogeny, but the other isogenies can be attacked similarly. The implemented attacks are based on the implementation from [40].

Results.

As suggested in [40], in order to increase the number of attempts by reducing the time required for a single run, we reduced the key space in CSIDH512 from 11^{74} to 3^2. Further, all required values, e.g. points of corresponding order, were calculated in advance, leading in total to a reduction from 15,721M to 115M clock cycles for a single run. Due to the reduced key space, private keys are of the form $S = (e_0, e_1)$, where $e_i \in [-1, 1]$. To obtain results for both cases (dummy and real), we performed experiments using different private keys. In the first case, the private key $S_1 = (-1, 1)$ consists of real isogenies only. Thus, attacks should not impact the computation of the shared secret. As expected, after 2,500 attempts, there is no faulty shared secret, achieving an accuracy of 100% (leading to an error probability $p_0 = 0$, as denoted in Figure 5.4). In the second case, however, the selected private key $S_2 = (0, 1)$ implies the calculation of a dummy isogeny since $e_0 = 0$. Hence, fault injections should lead to a faulty shared secret. Here, we achieved an accuracy of over 92% (denoted as p_1 in Figure 5.4). Table 5.6 shows the achieved results of the applied attacks in our setup. Hence,

Table 5.6: Results for CSIDH attacking the first isogeny

key	# of trials	faulty shared secret	accuracy
$S_1 = (-1, 1)$	2500	0.0%	100.0%
$S_2 = (0, 1)$	2500	92.4%	92.4%

based on these numbers, we assume an attacker can distinguish real from dummy isogenies with a single injection with high accuracy.

Since in dummy-based constant-time implementations of CSIDH (e.g., Meyer, Campos, and Reith (MCR) [136] or Onuki, Aikawa, Yamazaki, and Takagi (OAYT) [154]), the private key vector $(e_1, \dots, e_n)$ is sampled from an interval defined by a bound vector $\mathtt{m} = (m_1, m_2, \dots, m_n)$, the number of fault injections required to obtain the absolute value of a certain e_i strongly depends on the corresponding bound vector. More precisely, since the computation of a given degree ℓ_i occurs deterministically (real-then-dummy), the attacker performs a binary search through the corresponding m_i to identify the computation of the first dummy isogeny. Thus, the number of attacks required to obtain the absolute value of a certain e_i depends only on the corresponding bound m_i.

The achieved key space reductions are due to the fact that an attacker after a certain number of attacks knows the absolute values for the private key vector $(e_1, \dots, e_n)$. In the case of the OAYT implementation of CSIDH512 (where $-m_i \leq e_i \leq m_i, m_i = 5$ for $i = 0, \dots, 73$), our approach leads to a private key space reduction from 2^{256} to 2^{74} in the worst case ($e_i \neq 0$ for $i = 0, \dots, 73$) and to $2^{67.06}$ in the average case after at least $222 \cdot 4 = 888$ fault injections for a success rate over 99%. The remaining key space can be further reduced by a meet-in-the-middle approach [49] to about $2^{34.5}$ in the average case. For achieving a success rate over 99%, when attacking the MCR implementation (where $0 \leq e_i \leq m_i, m_i \in [1, 10]$ for $i = 0, \dots, 73$), at least $296 \cdot 4 = 1184$ injections are required for full key recovery (cf. Figure 5.4) since only positive values are allowed for the private key vector. Considering the running time of the non-optimised implementation of CSIDH512 of about 5 minutes for a single run in our setup, full key recovery would require about 98 hours in the case of the MCR implementation and about 74 hours to achieve the mentioned key space reduction in the case of the OAYT version.

Since recent works [35, 158] suggests that CSIDH-512 may not reach the post-quantum security as initially considered [49], some works recommend to increase the size of the CSIDH prime p [35, 158, 54]. However, from a classical perspective, since the classical security only depends on the size of the private key space, the number of prime factors ℓ_i remains unchanged. Thus, apart from the longer running time due to possibly larger prime factors, increasing the quantum security has no further influence on the effectiveness of the presented attack.

5.2.6 Countermeasures

In this section we discuss general countermeasures against safe-error attacks and then present concrete countermeasures for SIKE and CSIDH.

In safe-error attacks, a simple check of the final result before transmitting can still leak one bit. This can be easily seen in the attack on SIKE in Section 5.2.3.2. If the attacker successfully executes an attack, even if the result is checked for correctness, the implementation will leak one bit: either the algorithm fails or it returns an unusable result, or the induced error is overwritten, both of which represent a successful attack. This makes efficient generic countermeasures hard to design, as, for instance, simply repeating a calculation after a fault has been detected can be detected, too: an algorithm that suddenly takes twice as long shows that the attack was successful.

Using infective computation [97], a succesfully induced fault directly, i.e., without the necessity of checking, modifies the output value such that the faulty output does not allow to reveal secret values. In case of safe-error attacks, this is also not a solution, since any faulty output shows that the fault was successful. This is all an attacker needs to know in case of safe-error attacks.

An effective countermeasure consists in redundant computation with consistency check, i.e., calculating the susceptible operations repeatedly and then choose the value to be output by majority vote. However, this is costly, since, assuming that an attacker can realize a fault n times within a single computation of the algorithm, the susceptible operations have to be computed $2 \cdot n + 1$ times. Since second-order faults, i.e., two faults within one computation, are practical [32], this would require at least a fivefold repetition of the susceptible operations.

Another route, which is not in the hands of the implementer, is the selection of hardware the algorithm executes on. Hardware-based detection of fault attacks through, for instance, voltage sensing or intrusion detection, are possible ways of shutting down the execution - independent of the effect of the fault on the computation - before any information could have been leaked [192].

It is important to note that the attacks presented in this paper exploit secret-dependent memory access. Implementations and future optimizations should thus take special care to eliminate any such occurrence and treat them with the same rigour as secret-depending timings. This also extends to "branch-less" versions of algorithms, where, for instance, a pointer is swapped depending on the bit of a secret key; this does not remove the secret dependence of the underlying memory.

The discussion shows that to prevent safe-error attacks, the susceptible functions have to be adjusted, as in [188].

5.2.6.1 Securing SIKE

As explained in Section 5.2.3.1, by using a simultaneous double-and-add algorithm within **xDBLADD** [108], the particular M safe-error attack on SIKE can be prevented.

A possible countermeasure against the key recovery presented in Section 5.2.3.2 is to add an additional check to the **LADDER3PT** algorithm. The attack relies on skipping the `swap_points` method. Hence, a relatively inexpensive way of detecting an attack is to verify whether the swap actually took place. Thus, in each loop the implementation would save the current points, run the swap operation, and eventually check if the calculated mask had the intended effect.

Although the proposed countermeasure to conditional point swaps from [40] could be adapted to SIKE, the described approach (cf. [40], Section VI, Paragraph C, Point 1) represents no real countermeasure. An attack in the case where no swap takes place (decision bit = 0) does not lead to a false result (wrong point order), while attacking the conditional swap in the case of a swap (decision bit = 1) the order check of the resulting point should fail.

5.2.6.2 Securing CSIDH

Since the current CSIDH action algorithms branch on the secret key, it is a prime target for exploitation. One possible way of making attacks more difficult is shown in [123]. Here, LeGrow and Hutchinson show that using a binary decision vector to interleave the different ℓ_i-isogenies, an attacker has to do more than 8x as many attacks to gain the same amount of information.

Another approach is to choose an implementation that is dummy-free. So far however, dummy-free implementations have come at the cost of being twice as slow [51]. Further research might be able to close this performance gap and thereby completely eliminate attacks based on dummy isogenies.

Securing CSIDH against physical attacks is clearly difficult and care has to be taken to not accidentally enable another attack by fixing a specific vunerability. One such occurrence are dummy isogenies, introduced as timing attack countermeasures in [136], which allow an attacker to learn secret information through fault injections. Although in this specific situation using dummy isogenies might be reasonable as timing attacks are in general easier to carry out than fault injections, all implications a countermeasure can have have always to be considered so that implementers can consciously reason about trade-offs.

5.2.7 Conclusion

This work shows how safe-error attacks can be applied to recent isogeny-based cryptographic schemes. We presented four different attacks on the SIKE and CSIDH cryptosystems. It is important to note that the resilience of SIKE against the attack described in Section 5.2.3.1 solely depends on the structure of the actual implementation. As such, any further implemententations need to make sure to not introduce the possibility of this safe-error attack. We have shown how to practically realize two of these attacks and how to achieve full key recovery in a static key context on both SIKE and CSIDH.

We discussed that securing cryptosystems against safe-error attacks is non-trivial. This also partially explains why some of the attacks that we applied to isogeny-based cryptographic schemes have similarly been known in the ECC community for a long time, and yet have not been prevented in current implementations of SIKE and CSIDH. As safe-errors exploit differences of computation and memory access depending on the secret key, a simple check is not sufficient. It is equally important, that countermeasures against certain attacks do not open ways for further safe-error attacks [189]. This can be the case for example when implementing a simple consistency check, which might not trigger on all injections, thus inadvertently leaking data. The same holds true for constant-time implementations, which are designed to thwart timing attacks. The implementations of CSIDH that we attacked in this work are constant-time, but based on dummy isogenies, which enable our attack. CTIDH [12], a recent faster constant-time algorithm for CSIDH, is also vulnerable to safe-error attacks. In this case, the attacks should occur during the dummy operations within the *MatryoshkaIsogeny* (cf. [12], Section 5.2.2). Dummy-free implementations, which do also exist, are probably not vulnerable to the attacks presented in this paper; however, they are prone to timing attacks. Future research therefore needs to find a way to secure CSIDH at the same time against timing and safe-error attacks.

5.3 Zero-Value and Correlation Attacks on SIKE and CSIDH

This chapter is for all practical purposes identical to the paper *Patient Zero & Patient Six: Zero-Value and Correlation Attacks on CSIDH and SIKE* [43] authored jointly with Michael Meyer, Krijn Reijnders, and Marc Stöttinger, which was published at SAC 2022.

5.3.1 Introduction

Isogeny-based cryptography is a promising candidate for replacing pre-quantum schemes with practical quantum-resistant alternatives. In general, isogeny-based schemes feature very small key sizes, while suffering from running times that are at least an order of magnitude slower than e.g. lattice- or code-based schemes. Therefore, they present a viable option for applications that prioritize bandwidth over performance. SIKE [108], a key encapsulation mechanism (KEM) based on the key exchange SIDH [109], is the lone isogeny-based participant of the NIST post-quantum cryptography standardization process, and proceeded to the fourth round. In 2018, only after the NIST standardization process started, the key exchange scheme CSIDH was published [49]. Due to its commutative structure, a unique feature among the known post-quantum schemes, CSIDH allows for a non-interactive key exchange, which gained much attention among the research community. Together with its efficient key validation, which enables a static-static key setting, this makes CSIDH a promising candidate for a drop-in replacement of classical Diffie–Hellman-style schemes.

In this work, we focus on a side-channel attack against CSIDH and SIKE. We follow the main idea of [74], which reconstructs SIKE private keys through *zero-value* attacks. This attack approach tries to force zero values for some intermediate values of computations related to secret key bits. By recognizing these zero values via side-channel analysis (SCA), this allows an attacker to recover bits of the secret key. While *coordinate randomization* is an effective method to mitigate general *Differential Power Analysis* (DPA) and *Correlation Power Analysis* (CPA), it has no effect on zero values, such that forcing their occurrence bypasses this countermeasure, which is incorporated in SIKE [108].

While [74] focuses on forcing values connected to elliptic curve points becoming zero, we discuss the occurrence of zero values as curve parameters. This was first proposed in [120], yet [74] concludes that this idea is unlikely to be applicable in a realistic scenario, since curve representations in SIKE are such that they cannot produce a zero. In spite of this fact, we show that some curves in SIKE and CSIDH, as e.g. the zero curve, have a special correlation in these representations, which admits noticing their occurrence via side-channel analysis.

The secret isogeny computation in SIKE essentially consists of two phases: scalar multiplication and isogeny computation. In general, the first phase is believed to be more vulnerable to physical attacks, since private key bits are directly used there (see [61]). Our attack is the first passive side-channel attack that exclusively targets the second phase of the SIKE isogeny computation. Notably, countermeasures like coordinate/coefficient randomization [61] or the *CLN test* [64, 74] do not prevent our attack.

Our contributions. In this work, we present zero-value and correlation attacks against state-of-the-art implementations of CSIDH and SIKE. For CSIDH, we use the fact that the zero curve E_0, i.e., the Montgomery curve with coefficient $a = 0$, represents a valid curve. Thus, whenever a secret isogeny walk passes over this curve, this can be detected via side-channel analysis. We present an adaptive attack that recovers one bit of the secret key per round by forcing the target to walk over the zero curve.

Some implementations, like SQALE and SIKE, represent the zero curve without using zero values. Nevertheless, in such a case there is often (with probability 1/2 in SQALE and probability 1 in SIKE) a strong correlation between certain variables, which also occurs for the supersingular six curve E_6 with coefficient $a = 6$. Via CPA, we exploit this correlation to detect these curves.

Using these two approaches, we present a generic attack framework, and apply this attack to the state-of-the-art CSIDH implementations SQALE [54] and CTIDH [12] (Section 5.3.3), and to SIKE (Section 5.3.4). We explore the practical feasibility of our attacks (Section 5.3.5), simulations (Section 5.3.6), and different types of countermeasures (Section 5.3.7). Our code is available in the public domain:

`https://github.com/PaZeZeVaAt/simulation`

Related work. The analysis of physical attacks on isogeny-based schemes has only recently gained more attention, including both side-channel [74, 95, 120, 186, 193] and fault attacks [2, 40, 42, 94, 124, 177, 180]. Introduced for classical elliptic curve cryptography (ECC) in [4, 100, 105], zero-value attacks were adapted to SIKE in [74], which applies t-tests to determine zero values within power traces [165].

An approach to identify certain structures within traces, similar to the ones occurring in non-zero representations of the zero curve and six curve in our case, are correlation-enhanced power analysis collision attacks [142], such as [15] for ECC. This attack combines the concept of horizontal side-channel analysis [147] with correlation-enhanced power analysis collision attacks to extract leakage from a single trace.

We note that from a constructive perspective, our attack on SIKE is similar to the attack in [2]. However, our attack is a *passive* side-channel attack that is much easier to perform in practice compared to the elaborate fault injection required for [2].

5.3.2 Preliminaries

We briefly introduce mathematical background related to isogeny-based cryptography, and the schemes CSIDH [49] and SIKE [108]. For more mathematical details, we refer to [70].

Mathematical background. Let $\mathbb{F}_q$ with $q = p^k$ denote the finite field of order q, with a prime $p > 3$. Supersingular elliptic curves over $\mathbb{F}_q$ are characterized by the condition $\#E(\mathbb{F}_q) \equiv 1 \mod p$. Throughout this work, we will only encounter group orders that are multiples of 4, and hence elliptic curves E over $\mathbb{F}_q$ with $j(E) \in \mathbb{F}_q$ can be represented in Montgomery form:

$$E_a\colon y^2 = x^3 + ax^2 + x, \quad a \in \mathbb{F}_q. \tag{5.3}$$

Given two such elliptic curves E_a and $E_{a'}$, an isogeny is a morphism $\varphi : E_a \to E_{a'}$ such that $\mathcal{O}_{E_a} \mapsto \mathcal{O}_{E_{a'}}$ for the neutral elements of E_a and $E_{a'}$. In the context of isogeny-based cryptography, we are only interested in separable isogenies, which are characterized by their kernel (up to isomorphism): A finite subgroup $G \subset E_a(\overline{\mathbb{F}_q})$ defines a separable isogeny $\varphi : E_a \to E_a/G$ and vice versa. In such a case, the degree of φ is equal to the size of its kernel, $|G|$. For any isogeny $\varphi : E_a \to E_{a'}$, there is a unique isogeny $\hat{\varphi} : E_{a'} \to E_a$ such that $\hat{\varphi} \circ \varphi = [\deg(\varphi)]$ is the scalar point multiplication on E_a by $\deg(\varphi)$. We call $\hat{\varphi}$ the dual isogeny. Two elliptic curves E_a and $E_{a'}$ over $\mathbb{F}_q$ are isogenous, i.e., there exists an isogeny between them, if and only if $\#E_a(\mathbb{F}_q) = \#E_{a'}(\mathbb{F}_q)$.

5.3.2.1 CSIDH

In the context of CSIDH, we choose p of the form $p+1 = h \cdot \prod_{i=1}^{n} \ell_i$ and work with supersingular elliptic curves over $\mathbb{F}_p$. Each ℓ_i is a small odd prime, and h is a suitable cofactor to ensure p is prime, with the additional requirement that $4 \mid h$. This ensures that the group order $p+1$ is a multiple of 4, and any such supersingular elliptic curve can be uniquely represented in Montgomery form [49].

We are interested in specific isogenies $\mathfrak{l}_i$ (of degree ℓ_i) that are defined by the kernel $G = E[\ell_i] \cap E[\pi - 1]$, where π denotes the Frobenius endomorphism, i.e., $\mathbb{F}_p$-rational points that have ℓ_i-torsion. As E_a has $p+1$ points over $\mathbb{F}_p$, we get $E_a(\mathbb{F}_p) \cong \mathbb{Z}_h \times \prod_{i=1}^{n} \mathbb{Z}_{\ell_i}$. This implies there are ℓ_i of such points $P \in E[\ell_i] \cap E[\pi - 1]$, and $\ell_i - 1$ of these (all but the point $\mathcal{O}_{E_a}$) will generate G. The codomain $E_{a'}$ of such an isogeny is again supersingular and so $|E_{a'}(\mathbb{F}_p)| = p+1$, which implies $\mathfrak{l}_i$ can also be applied to $E_{a'}$. This implies a group action of the elements $\mathfrak{l}_i$ on the supersingular curves E_a over $\mathbb{F}_p$, which we denote by $[\mathfrak{l}_i] * E_a$. In particular, this group action is commutative: $[\mathfrak{l}_i \mathfrak{l}_j] * E_a \cong [\mathfrak{l}_i] * [\mathfrak{l}_j] * E_a \cong [\mathfrak{l}_j] * [\mathfrak{l}_i] * E_a \cong [\mathfrak{l}_j \mathfrak{l}_i] * E_a$. The dual isogeny associated to $\mathfrak{l}$ is denoted as $\mathfrak{l}^{-1}$, and is defined by the kernel $G = E[\ell_i] \cap E[\pi + 1]$.

The CSIDH scheme. The CSIDH scheme is based on the group action as described above: We apply each of the n different $\mathfrak{l}_i^{\pm 1}$ a number of times to a given curve E_a, and we denote this number by e_i. Hence, the secret key is some vector of n integers $(e_1, \dots, e_n)$ defining an element $\mathfrak{a} = \prod_{i=1}^{n} \mathfrak{l}_i^{e_i}$ which we can apply to supersingular curves E_a over $\mathbb{F}_p$. There is some variation between different proposals on where e_i is chosen from: The original proposal of CSIDH-512 picks $e_i \in \{-5, \dots, 5\}$, but one can also vary the bound m_i per e_i. The key space is of size $\prod(2m_i + 1)$. For the original CSIDH-512 proposal with $m_i = 5$ and $n = 74$, this gives roughly size 2^{256}.

The public key is the supersingular curve E_a corresponding to applying the secret key $\mathfrak{a}$ to the publicly known starting curve $E_0 : y^2 = x^3 + x$:

$$E_a = \mathfrak{a} * E_0 = \mathfrak{l}_1^{e_1} * \cdots * \mathfrak{l}_n^{e_n} * E_0. \tag{5.4}$$

To derive a shared secret between Alice and Bob with secret keys $\mathfrak{a}$ and $\mathfrak{b}$ and given public keys $E_a = \mathfrak{a} * E_0$ and $E_b = \mathfrak{b} * E_0$, Alice simply computes $E_{ab} = \mathfrak{a} * E_b$ and Bob computes $E_{ba} = \mathfrak{b} * E_a$. From the commutativity of the group action, we get $E_{ab} \cong E_{ba}$.

Security of CSIDH. The classical security relies mostly on the size of the keyspace $\prod(2m_i + 1)$, but the quantum security of CSIDH is heavily dependent on the size of the group generated by these elements $\mathfrak{l}_i$. It is heuristically assumed that the $\mathfrak{l}_i$ generate a group of size approximately $\sqrt{p}$. While the original CSIDH proposal considered a 512-bit prime p sufficient for NIST security level 1 [49], its exact quantum security is debated [23, 158, 35, 54]. For instance, [54] claims that 4096-bit primes are required for level 1 security. Note that the key space is not required to cover the full group of size roughly $\sqrt{p}$, but can be chosen as a large enough subset, except for particularly bad choices like subgroups. At larger prime sizes, the number n of small primes ℓ_i grows, and therefore it becomes natural to pick secret key vectors from $\{-1, 0, 1\}^n$ resp. $\{-1, 1\}^n$ for primes sizes of at least 1792 resp. 2048 bits. This allows for a large enough key space for classical security, while increasing p for increased quantum security.

We note that the exact quantum security of CSIDH remains unclear, and thus work on efficient and secure implementations for both smaller and larger parameters continues to appear, e.g. in [12, 54].

Constant-time implementations. CSIDH is inherently difficult to implement in constant time, as this requires that the timing of the execution is independent of the respective secret key $(e_1, \dots, e_n)$. However, picking a secret key vector $(e_1, \dots, e_n)$ translates to the computation of $|e_i|$ isogenies of degree ℓ_i, which directly affects the timing of the group action evaluation. One way to mitigate this timing leakage is by using dummy isogenies: We can keep the total number of isogenies per degree constant by computing m_i isogenies of degree ℓ_i, but discarding the results of $m_i - |e_i|$ of these, effectively making them dummy computations [137, 136]. Several optimizations and different techniques have been proposed in the literature [154, 51, 56].

The latest and currently most efficient variant of constant-time implementations of CSIDH is CTIDH [12]. In contrast to sampling private key vectors such that $e_i \in \{-m_i, \dots, m_i\}$, CTIDH uses a different key space that exploits the approach of batching the primes ℓ_i. We define *batches* $B_1, \dots, B_N$ of consecutive primes of lengths $n_1, \dots, n_N$, i.e., $B_1 = (\ell_{1,1}, \dots, \ell_{1,n_1}) = (\ell_1, \dots, \ell_{n_1})$, $B_2 = (\ell_{2,1}, \dots, \ell_{2,n_2}) = (\ell_{n_1+1}, \dots, \ell_{n_1+n_2})$, et cetera. We write $e_{i,j}$ for the (secret) coefficient associated to $\ell_{i,j}$. Instead of defining bounds m_i for each individual ℓ_i so that $|e_i| \le m_i$, CTIDH uses bounds M_i for the batch B_i, i.e., we compute at most M_i isogenies of those degrees that are contained in B_i. That is, the key sampling requires $|e_{i,1}| + \dots + |e_{i,n_i}| \le M_i$. CTIDH then adapts the CSIDH algorithm such that the distribution of the M_i isogenies among degrees of batch B_i does not leak through the timing channel. Among other techniques, this involves Matryoshka isogenies, first introduced in [23], that perform the exact same sequence of instructions independent of its isogeny degree $\ell_{i,j} \in B_i$.

The main advantage of CTIDH is the ambiguity of the isogeny computations: From a time-channel perspective, a Matryoshka isogeny for B_i could be an $\ell_{i,j}$-isogeny for any $\ell_{i,j} \in B_i$. Thus, in comparison to the previous CSIDH algorithms, CTIDH covers the same key space size in fewer isogenies. For instance, the previously fastest implementation of CSIDH-512 required 431 isogenies in total [3] (including dummies), whereas CTIDH [12] requires only 208 isogenies (including dummies) for the same key space size. This leads to an almost twofold speedup.

Representation of Montgomery coefficients. To decrease computational costs by avoiding costly inversions, the curve E_a is almost always represented using *projective* coordinates for $a \in \mathbb{F}_p$. The following two are used most in current CSIDH-based implementations:

- the Montgomery form $(A : C)$, such that $a = A/C$, with C non-zero,
- and the alternative Montgomery form $(A+2C : 4C)$, such that $a = A/C$, with C non-zero.

The alternative Montgomery form is most common, as it is used in projective scalar point multiplication formulas. Hence, in most state-of-the-art implementations of CSIDH-based systems, the Montgomery coefficient a is mapped to alternative Montgomery form and remains in this form until the end, where it is mapped back to affine form for the public key resp.

shared secret (e.g., in SQALE [54]). CTIDH [12] switches between both representations after each isogeny, and maps back to affine $a = A/C$ at the end. For most values of $(A : C)$ and $(A+2C : 4C)$, $a = A/C$ represents either an ordinary or a supersingular curve. The exceptions are $C = 0$, which represents no algebraic object, and $A = \pm 2C$, which represents the singular curves $E_{\pm 2}$. Specifically the supersingular zero curve E_0 is represented as $(0 : C)$ in Montgomery form and $(2C : 4C)$ in alternative Montgomery form, where $C \in \mathbb{F}_p$ can be any non-zero value.

Isogeny computation in projective form. When using projective representations to compute isogenies with domain E_a where a is represented as $(A : C)$, most implementations use projectivized versions of Vélu's formulas, described in [184, 141, 19]. To compute the action of $\mathfrak{l}_i^{\pm 1}$ on E_a, one finds a point P of order ℓ_i on E_a and computes the x-coordinates of the points $\{P, [2]P, \dots, [\frac{\ell-1}{2}]P\}$. Let $(X_k : Z_k)$ denote the x-coordinate of $[k]P$ in projective form. Then, the projective Montgomery coefficient $(A' : C')$ of $E_{a'} = \mathfrak{l}_i \star E_a$ using Montgomery form $(A : C)$ is computed by

$$B_z = \prod_{k=1}^{\frac{\ell-1}{2}} Z_k, \quad A' = (A+2C)^{\ell} \cdot B_z^8, \tag{5.5}$$

$$B_x = \prod_{k=1}^{\frac{\ell-1}{2}} X_k, \quad C' = (A-2C)^{\ell} \cdot B_x^8, \tag{5.6}$$

and when using alternative Montgomery form $(\alpha : \beta) = (A+2C : 4C)$ by

$$B_z = \prod_{k=1}^{\frac{\ell-1}{2}} Z_k, \quad \alpha' = \alpha^{\ell} \cdot B_z^8, \tag{5.7}$$

$$B_x = \prod_{k=1}^{\frac{\ell-1}{2}} X_k, \quad \beta' = \alpha' - (\alpha-\beta)^{\ell} \cdot B_x^8, \tag{5.8}$$

where $(\alpha' : \beta')$ represents $E_{a'}$ in alternative Montgomery form. Note that the values $(A+2C)$ in (5.5), $(A-2C)$ in (5.6), α in (5.7) and $(\alpha-\beta)$ in (5.8) are never zero: In all cases, this implies $A/C = \pm 2$, i.e., the singular curves $E_{\pm 2}$.

Remark 7. *So far, we know of no deterministic implementations based on the class group action. This is because in order to perform the isogenies, all current implementations sample a random point P on the curve and compute the scalar multiple of P required to perform isogenies. The projective coordinates $(X_k : Z_k)$ are then non-deterministic, and hence the output of Equations (5.5) to (5.8) is non-deterministic. This implies that the representation of a as $(A : C)$ or $(A+2C : 4C)$ is non-deterministic after the first isogeny.*

5.3.2.2 SIKE

In SIKE, we pick a prime of the form $p = 2^{e_A} \cdot 3^{e_B} - 1$ such that $2^{e_A} \approx 3^{e_B}$, and work with supersingular elliptic curves over $\mathbb{F}_{p^2}$ in Montgomery form. We choose to work with curves such that $\#E_a(\mathbb{F}_{p^2}) = (p+1)^2$, and we have $E_a(\mathbb{F}_{p^2}) \cong \mathbb{Z}^2_{2^{e_A}} \times \mathbb{Z}^2_{3^{e_B}}$ for these curves. Thus, the full 2^{e_A}- and 3^{e_B}-torsion subgroups lie in $E_a(\mathbb{F}_{p^2})$. Any point R_A of order 2^{e_A} then uniquely (up to isomorphism) determines a 2^{e_A}-isogeny and codomain curve $E_{a'} = E_a/\langle R_A \rangle$ with kernel $\langle R_A \rangle$. For choosing an appropriate point, the SIKE setup defines basis points P_A and Q_A of the 2^{e_A}-torsion of the public starting curve. Picking an integer $\mathsf{sk}_A \in [0, 2^{e_A} - 1]$ and computing $R_A = P_A + [\mathsf{sk}_A]Q_A$ then results in choosing such a kernel generator R_A of order 2^{e_A}.

In practice, such a 2^{e_A}-isogeny is computed as a sequence of 2-isogenies of length e_A. This can be interpreted as a sequence of steps through a graph: For a prime ℓ with $p \nmid \ell$, the ℓ-isogeny graph consists of vertices that represent (j-invariants of) elliptic curves, and edges representing ℓ-isogenies. Due to the existence of dual isogenies, edges are undirected. For supersingular curves, this graph is an $(\ell+1)$-regular expander graph and contains approximately $p/12$ vertices. Hence, a sequence of 2-isogenies of length e_A corresponds to a walk of length e_A through the 2-isogeny graph. An analogous discussion applies to the case of 3^{e_B}-isogenies. Note that for reasons of efficiency, we often combine two 2-isogeny steps into one 4-isogeny. The secret keys sk_A, sk_B can be decomposed as

$$\mathsf{sk}_A = \sum_{i=0}^{e_2-1} \mathsf{sk}_i \cdot 2^i \quad \mathsf{sk}_i \in \{0,1\}, \qquad \mathsf{sk}_B = \sum_{i=0}^{e_3-1} \mathsf{sk}_i \cdot 3^i \quad \mathsf{sk}_i \in \{0,1,2\}.$$

We refer to these sk_i as the *bits* resp. the *trits* of the secret key sk_A resp. sk_B. For a given sk, we use $\mathsf{sk}_{<k}$ to represent the key up to the k-th bit/trit sk_{k-1}.

The SIKE scheme. The main idea behind SIDH and SIKE is to use secret isogenies to set up a key exchange scheme resp. key encapsulation mechanism. SIDH fixes E_6 as starting curve, and torsion basis points P_A, Q_A and P_B, Q_B. It uses the following subroutines:

- KeyGen_A samples a secret key $\mathsf{sk}_A \in [0, 2^{e_A} - 1]$, computes $R_A = P_A + [\mathsf{sk}_A]Q_A$, and the secret isogeny $\phi_A : E_6 \to E_6/\langle R_A \rangle$. It outputs the key pair $(\mathsf{sk}_A, \mathsf{pk}_A)$, where $\mathsf{pk}_A = (\phi_A(P_B), \phi_A(Q_B), \phi_A(Q_B - P_B))$.

- KeyGen_B proceeds analogously with swapped indices A and B. The public key is $\mathsf{pk}_B = (\phi_B(P_A), \phi_B(Q_A), \phi_B(Q_A - P_A))$.

- $\mathtt{Derive}_A$ takes as input $(\mathsf{sk}_A, \mathsf{pk}_B) = (S_A, T_A, T_A - S_A)$. It computes the starting curve E_B from the points in pk_B, the secret point $R'_A = S_A + [\mathsf{sk}_A]T_A$, and the isogeny $\phi'_A : E_B \to E_B/\langle R'_A \rangle$.

- $\texttt{Derive}_B$ proceeds analogously with input $(\mathsf{sk}_B, \mathsf{pk}_A)$, and computes the codomain curve $E_A/\langle R'_B \rangle$.

When running this key exchange, both parties arrive at a curve (isomorphic to) $E_6/\langle R_A, R_B \rangle$, and (a hash of) its j-variant can serve as a shared secret.

SIKE uses the SIDH subroutines KeyGen and $\texttt{Derive}$ to construct three algorithms KeyGen, Encaps, and Decaps. Furthermore, we define h and h' to be cryptographic hash functions.

- KeyGen runs KeyGen_B to generate a key pair $(\mathsf{sk}, \mathsf{pk})$. This key pair can be chosen to be static.
- Encaps picks a message m, runs KeyGen_A to generate an ephemeral key pair $(\texttt{ek}, c)$ with $\texttt{ek} = h(\mathsf{pk}, m)$, and computes the shared secret s through $\texttt{Derive}_A(\texttt{ek}, \mathsf{pk})$. It encapsulates m as a ciphertext $\texttt{ct} = (c, h'(s) \oplus m)$.
- Decaps receives a ciphertext (c_0, c_1), recomputes the shared secret s' from $\texttt{Derive}_B(\mathsf{sk}, c_0)$, and the message $m' \leftarrow c_1 \oplus h'(s')$. It recomputes $\texttt{ek}' = h(\mathsf{pk}, m')$ and checks if $c_0 = c$ by running $\texttt{Derive}_A(\texttt{ek}', \mathsf{pk})$. Passing this check guarantees that the ciphertext has been generated honestly, and $m' = m$ can be used to set up a session key.

Representation of Montgomery coefficients. As in CSIDH, the curve E_a is almost always represented using projective coordinates, with the caveat that $a \in \mathbb{F}_{p^2}$. The following two representations are used throughout SIKE computations, although in different subroutines.

- The alternative Montgomery form $(A + 2C : 4C)$, such that $a = A/C$ with C non-zero. This representation is used for Alice's computations as it is the most efficient for computing 2-isogenies. It is often written as $(A_{24}^+ : C_{24})$ with $A_{24}^+ = A + 2C$ and $C_{24} = 4C$ so that $a = 2(2A_{24}^+ - C_{24})/C_{24}$.
- The form $(A + 2C : A - 2C)$, such that $a = A/C$, with C non-zero. This representation is used for Bob's computations as it is the most efficient for computing 3-isogenies. It is often written as $(A_{24}^+ : A_{24}^-)$ with $A_{24}^+ = A + 2C$ and $A_{24}^- = A - 2C$ so that $a = 2(A_{24}^+ + A_{24}^-)/(A_{24}^+ - A_{24}^-)$.

Note that the values A, C, A_{24}^+, A_{24}^- and C_{24} are in $\mathbb{F}_{p^2}$. When necessary, we write them as $\alpha + \beta i$ with $\alpha, \beta \in \mathbb{F}_p$ and $i^2 = -1$. Equal to CSIDH, both forms represent either an ordinary or a supersingular curve, with the exceptions $C = 0$, which represents no algebraic object, and $A = \pm 2C$, which represents the singular curves $E_{\pm 2}$.

For the rest of the paper, we are interested in representations of the supersingular six curve E_6. Fortunately, E_6 is represented in *both* forms as $(8C : 4C)$, with $C = \alpha + \beta i \in \mathbb{F}_{p^2}$ any non-zero element. For the goal of the paper, this means that the analysis is similar for both forms.

Isogeny computation in projective form. SIKE uses the above projective representations to compute the codomain $E_{\tilde{a}}$ of a 3- or 4-isogeny $\phi : E_a \to E_{\tilde{a}}$.

4-isogeny. Given a point P of order 4 on E_a with x-coordinate $x(P) = (X : Z)$, the codomain $E_{\tilde{a}} = E_a/\langle P \rangle$ with $\tilde{a}$ represented by $(\tilde{A}^+_{24} : \tilde{C}_{24})$ is computed by

$$\tilde{A}^+_{24} = 4 \cdot X^4, \qquad \tilde{C}_{24} = 4 \cdot Z^4. \tag{5.9}$$

3-isogeny. Given a point P of order 3 on E_a with x-coordinate $x(P) = (X : Z)$, the codomain $E_{\tilde{a}} = E_a/\langle P \rangle$ with $\tilde{a}$ represented by $(\tilde{A}^+_{24} : \tilde{A}^-_{24})$ is computed by

$$\tilde{A}^+_{24} = (3X + Z)^3 \cdot (X - Z), \quad \tilde{A}^-_{24} = (3X - Z)^3 \cdot (X + Z). \tag{5.10}$$

5.3.3 Recovering CSIDH keys with E_0 side-channel leakage

In this section, we explore how side-channel information can leak information on secret isogeny walks. As shown in [74], it is possible to detect zero values in isogeny computations using side-channel information. In Section 5.3.3.1, we specifically explore how both representations of the zero curve E_0, i.e. $(0 : C)$ and $(2C : 4C)$, leak secret information, even though the value $C \in \mathbb{F}_p$ is assumed to be a uniformly random non-zero value. As E_0 is always a valid supersingular $\mathbb{F}_p$-curve in CSIDH, we can always construct a walk that potentially passes over E_0. This allows us to describe a generic approach to leak a given bit of information of the secret isogeny walk, hence, a general attack on the class group action as introduced in CSIDH. We apply this attack in more detail to the two current state-of-the-art cryptosystems based on this class group action: SQALE in Section 5.3.3.2 and CTIDH in Section 5.3.3.3. We discuss their practical feasibility in Section 5.3.5 and simulate these attacks in Section 5.3.6. We note that our attack applies to all variants of CSIDH that we know of, e.g. from [49, 51].

Throughout this work, we assume a static-key setting, i.e., that a long-term secret key $\mathfrak{a}$ is used, and that the attacker can repeatedly trigger key exchange executions on the target device using public key curves of their choice. Formally, this means that we adaptively feed curves E_{PK} and get side-channel information on the computations $\mathfrak{a} * E_{\mathsf{PK}}$. We exploit this information to reveal $\mathfrak{a}$ bit by bit.

5.3.3.1 Discovering a bit of information on a secret isogeny walk

Detecting E_0 in Montgomery form. As described in Remark 7, the representation of the Montgomery coefficient as $(A : C)$ or $(A + 2C : 4C)$ is non-deterministic after the first isogeny, so they effectively contain random $\mathbb{F}_p$-values, representing the affine Montgomery coefficient a. This makes it hard to get any information on E_a using side channels. However, in Montgomery form the curve E_0 is special: It is simply represented by $(0 : C)$ for some $C \in \mathbb{F}_p$. We define such a representation containing a zero a *zero-value representation.*

Definition 21. *Let E_a be an elliptic curve over $\mathbb{F}_p$. A* zero-value representation *is a representation of the Montgomery coefficient a in projective coordinates $(\alpha : \beta)$ such that either $\alpha = 0$ or $\beta = 0$.*

Clearly, a representation of E_0 in Montgomery form must be a zero-value representation. As known for ECC and SIKE, an attacker can observe zero-value representations in several different ways using side-channel analysis [74]. We will expand on this in Section 5.3.5 to show that E_0 leaks secret information in implementations that use Montgomery form.

Detecting E_0 in alternative Montgomery form. Using the alternative Montgomery form, no non-singular curve has a zero-value representation, as $(A + 2C : 4C)$ can only be zero for $A = -2C$ corresponding to $a = -2$, which represents the singular curve E_{-2}. Thus, the alternative Montgomery form avoids the side-channel attack described above. Nevertheless, the representation of E_0 is still unusual: Whenever $2C$ is smaller than $p/2$, doubling $2C$ does not require a modular reduction, and hence the bit representation of $4C$ is precisely a bit shift of $2C$ by one bit to the left. Such strongly correlated values can be observed in several ways using side-channel analysis, as we detail later in Section 5.3.5.

Definition 22. *Let E_a be an elliptic curve over $\mathbb{F}_p$. A* strongly-correlated representation *is a representation of the Montgomery coefficient a in projective coordinates $(\alpha : \beta)$ such that $(\alpha, \beta) \in \mathbb{F}_p^2$ is distinguishable from random pairs $(\gamma, \delta) \in \mathbb{F}_p^2$.*

For E_0, for any non-zero value C with $2C \leq p/2$, the representation in alternative Montgomery form by $(2C : 4C)$ is a strongly-correlated representation. As C is effectively random during the computation of the class group action, in roughly 50% of the cases where we pass over E_0, the representation is strongly correlated. For random values of a, the values of $(A + 2C : 4C)$ are indistinguishable from random $(\gamma : \delta)$, and so an attacker can differentiate E_0 from such curves. From this, an attacker only needs a few traces to determine accurately whether a walk passes over E_0 or not, as discussed in Section 5.3.5.

Remark 8. *Other curves have strongly-correlated representations too, e.g., the curve E_6 requires $A = 6C$ which gives $(8C : 4C)$ with $C \in \mathbb{F}_p$ random and non-zero, and so E_6 can be detected in precisely the same way as E_0. For simplicity, we focus on the zero curve in our CSIDH attack. We note that analyzing this attack to any curve with strongly-correlated representations is of independent interest for CSIDH and other isogeny-based schemes (such as SIKE).*

Remark 9. *In the case where $2C$ is larger than $p/2$, the modular reduction by p decreases the correlation between $2C$ and $4C$ significantly, which is why we disregard these cases. However, a modular reduction does not affect all bits, and so this correlation remains for unaffected bits. Especially for primes with large cofactor 2^k in $p+1$, or primes close to a power of 2, the correlation between unaffected bits should be exploitable. For the primes used in the CSIDH instances in this work, this effect is negligible. However, the primes used in SIDH and SIKE do have this form and we exploit this in Section 5.3.4.*

The idea is now to detect E_0 in a certain step k of the computation $\mathfrak{a} * E_{\mathsf{PK}}$. In order to ensure that this happens the computation needs to be performed in a known order of isogeny steps $E \to \mathfrak{l}^{(k)} * E$. In general, by the way how isogenies are computed, such a step can fail with a certain probability. The following definition takes this into account.

Definition 23. *Let $\mathfrak{a}$ be a secret isogeny walk. An* ordered evaluation *of $\mathfrak{a} * E$ is an evaluation in a fixed order*

$$\mathfrak{l}^{(n)} * \ldots * \mathfrak{l}^{(1)} * E$$

of n steps, assuming that no step fails. We write $\mathfrak{a}_k * E$ for the first k steps of of such an evaluation,

$$\mathfrak{l}^{(k)} * \ldots * \mathfrak{l}^{(1)} * E.$$

We define $p_{\mathfrak{a}}$ resp. $p_{\mathfrak{a}_k}$ as the probability that $\mathfrak{a}$ resp. $\mathfrak{a}_k$ is evaluated without failed steps.

Generic approach to discover isogeny walks using E_0. Given the ability to detect E_0 in a walk for both the Montgomery form and the alternative Montgomery form, we sketch the following approach to discover bits of a secret isogeny walk $\mathfrak{a}$ that has an ordered evaluation. Assuming we know the first $k-1$ steps $\mathfrak{l}^{(k-1)} * \ldots * \mathfrak{l}^{(1)}$ in the secret isogeny walk $\mathfrak{a}$, denoted by $\mathfrak{a}_{k-1}$, we want to see if the k-th step $\mathfrak{l}^{(k)}$ equals $\mathfrak{l}_i$ or $\mathfrak{l}_i^{-1}$ for some i. We compute $E_a = \mathfrak{l}_i^{-1} * E_0$ and $E_{a'} = \mathfrak{l}_i^{-1} * E_a$, and as a public key we use $E_{\mathsf{PK}} = \mathfrak{a}_{k-1}^{-1} * E_a$. Then, when applying the secret walk $\mathfrak{a}$ to E_{PK}, the k-th step either goes over E_0 or over $E_{a'}$. From side-channel information, we observe if the k-th step

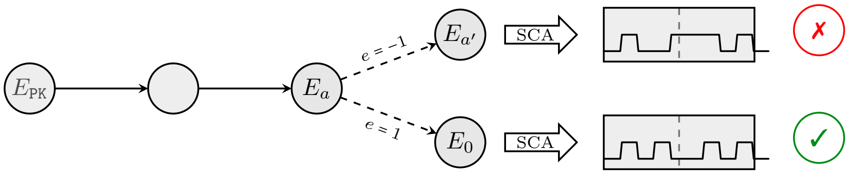


Figure 5.5: Generic approach to discover secret bits using side-channel information.

applies $\mathfrak{l}_i^e = \mathfrak{l}_i^1$ or $\mathfrak{l}_i^e = \mathfrak{l}_i^{-1}$, and set $\mathfrak{l}^{(k)} = \mathfrak{l}_i^e$, as shown in Figure 5.5. Then we repeat with $\mathfrak{a}_k = \mathfrak{l}_i^e \cdot \mathfrak{a}_{k-1}$.

If E_0 is not detected in the above setting, i.e. $e = -1$, we can confirm this by an additional measurement: We compute $\tilde{E_a} = \mathfrak{l} * E_0$ and $\tilde{E_{a'}} = \mathfrak{l} * \tilde{E_a}$, and use $\tilde{E}_{\text{PK}} = \mathfrak{a}_{k-1}^{-1} * \tilde{E_a}$ as public key. If $e = -1$, the isogeny walk now passes over E_0, which can be recognized via side-channel analysis. More formally, we get:

Lemma 2. *Let $\mathfrak{a}$ be any isogeny walk of the form $\mathfrak{a} = \prod \mathfrak{l}_i^{e_i}$. Assume the evaluation of $\mathfrak{a}$ is an ordered evaluation. Then, there exists a supersingular curve E_{PK} over $\mathbb{F}_p$ such that $\mathfrak{a} * E_{\text{PK}}$ passes over E_0 in the k-th step.*

This generic approach has a nice advantage: If one detects the k-th step to walk over E_0, this confirms all previous steps were guessed correctly. In other words, guessing wrongly in a certain step will be noticed in the next step: Denote a wrong guess by $\mathfrak{a}_k^{\text{wrong}} = \mathfrak{l}^{-e} \cdot \mathfrak{a}_{k-1}$. The attacker computes E_a from E_0 so that $\mathfrak{l}' * E_a = E_0$ and gives the target E_{PK} such that $\mathfrak{a}_k^{\text{wrong}} * E_{\text{PK}} = E_a$. Due to the wrong guess, neither $e = 1$ nor $e = -1$ lead to E_0, as the *actual* secret walk $\mathfrak{a}$ leads to $E_{a'} = \mathfrak{a}_k * E_{\text{PK}}$, and the case $e = 1$ leads to $E_{\neg 0} = \mathfrak{l}' * E_{a'} = \mathfrak{l}^{-2e} * E_0$, as shown in Figure 5.6.

Remark 10. *Note that E_{PK} given by Lemma 2 is a valid CSIDH public key, so public key validation (see [49]) does not prevent this attack.*

Probability of a walk passing over E_0. Due to the probabilistic nature of the computation of the class group action, not every evaluation $\mathfrak{a} * E_{\text{PK}}$ passes over E_0 in the k-th step: One of the steps $\mathfrak{l}^{(j)}$ for $1 \leq j \leq k-1$ can fail with probability $1/\ell^{(j)}$, and if so, the k-th step passes over a different curve. With E_{PK} as given by Lemma 2, the probability that an ordered evaluation $\mathfrak{a} * E_{\text{PK}}$ passes over E_0 is then described by $p_{\mathfrak{a}_k}$, which we compute in Lemma 3.

Lemma 3. *Let $\mathfrak{a}$ be an isogeny walk computed as an ordered evaluation $\mathfrak{l}^{(n)} * \ldots * \mathfrak{l}^{(1)} * E_{\text{PK}}$. Then $p_{\mathfrak{a}_k}$, the probability that the first k isogenies succeed, is*

$$p_{\mathfrak{a}_k} := \prod_{j=1}^{k} \frac{\ell^{(j)} - 1}{\ell^{(j)}}$$

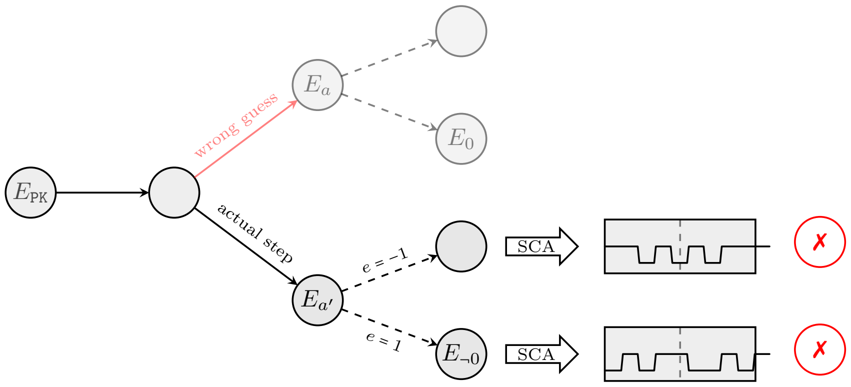


Figure 5.6: Due to a wrong guess of the isogeny path $\mathfrak{a}_k$, an attacker mis-computes E_{PK} and the actual walk does not pass over E_0.

where $\ell^{(j)}$ is the degree of the isogeny $\mathfrak{l}^{(j)}$ in the j-th step.

As $p_{\mathfrak{a}_k}$ describes the chance that we pass over E_0 in the k-th step, $1/p_{\mathfrak{a}_k}$ gives us the estimated number of measurements of $\mathfrak{a} * E_{\mathsf{PK}}$ we need in order to pass over E_0 in step k. We apply this more concretely in Sections 5.3.3.2 and 5.3.3.3.

Remark 11. *Instead of learning bit by bit starting from the beginning of the secret isogeny walk, we can also start at the end of the walk. To do so, we use the twist E_{-t} of the target's public key E_t, for which $\mathfrak{a} * E_{-t} = E_0$. As for the generic attack, we feed $E_{\mathsf{PK}} = \mathfrak{l}^{-1} * E_{-t}$ and $\tilde{E}_{\mathsf{PK}} = \mathfrak{l} * E_{-t}$. The computation then passes over E_0 in the* last *step instead of the* first*. This approach requires the same probability $p_{\mathfrak{a}_k}$ to recover the k-th bit, but assumes knowledge of all bits after k instead of before. Hence, we can discover starting and ending bits of $\mathfrak{a}$ in parallel.*

5.3.3.2 Recovering secret keys in SQALE

SQALE [54] is the most recent and most efficient constant-time implementation of CSIDH for large parameters, featuring prime sizes between 1024 and 9216 bit. In this section, we explain how the attack from Section 5.3.3.1 can be applied to SQALE, leading to a full key recovery. For concreteness, we focus on SQALE-2048, which uses parameters $n = 231$ and secret exponents $e_i \in \{-1, 1\}$ for $1 \leq i \leq 221$. The ℓ_i with $i > 221$ are not used in the group action.

Algorithmic description of SQALE. Given a starting curve E_A, the SQALE implementation computes the group action in the following way:

- Sample random points $P_+ \in E_A[\pi - 1]$ and $P_- \in E_A[\pi + 1]$, and set $E \leftarrow E_A$.
- Iterate through $i \in \{1, \dots, n\}$ in ascending order, and attempt to compute $\phi : E \to \mathfrak{l}_i^{e_i} * E$ using P_+ resp. P_-. Push both points through each ϕ.
- In case of point rejections, sample fresh points and attempt to compute the corresponding isogenies, until all $\mathfrak{l}_i^{e_i}$ have been applied.

In order to speed up computations, SQALE additionally pushes intermediate points through isogenies, which saves computational effort in following steps [56]. However, the exact design of the computational strategy inside CSIDH is not relevant for our attack. Using the above description, we sketch the adaptive attack on SQALE-2048 to recover the secret key bit by bit. In case of no point rejections, the order of steps in which $\mathfrak{a} * E_{\mathsf{PK}}$ is computed in SQALE is deterministic, and thus we can immediately apply Lemmas 2 and 3:

Corollary 7. *If no point rejections occur, the computation $\mathfrak{a} * E_{\mathsf{PK}}$ in SQALE is an ordered evaluation with*

$$\mathfrak{l}^{(n)} * \dots * \mathfrak{l}^{(1)} * E_{\mathsf{PK}} = \mathfrak{l}_{221}^{e_{221}} * \dots * \mathfrak{l}_1^{e_1} * E_{\mathsf{PK}}.$$

Hence,

$$p_{\mathfrak{a}_k} = \prod_{i=1}^{k} \frac{\ell_i - 1}{\ell_i}.$$

SQALE uses coefficients in alternative Montgomery form $(A + 2C : 4C)$, so that passing over the curve E_0 can be detected as described in Section 5.3.3.1.

Recovering the k-th bit. Recovering the k-th bit of a SQALE secret key works exactly as described in Figure 5.5, as in a successful run SQALE performs each step $\mathfrak{l}_i^{\pm 1}$ in ascending order. Thus, the k-th step, in a run where the first k steps succeed, computes $E \to \mathfrak{l}_k^{\pm 1} * E$. For the attack, we assume knowledge of the first $k - 1$ bits of the secret to produce public keys E_{PK} resp. $\tilde{E}_{\mathsf{PK}}$ that lead the target through E_0 via an application of $\mathfrak{l}_k^{-1}$ resp. $\mathfrak{l}_k$, as given by Lemma 2. For one of these cases, with probability $p_{\mathfrak{a}_k}$ (Lemma 3), the target passes over E_0 on the k-th step, and we learn the k-th secret bit e_k from side-channel information.

As k increases, $p_{\mathfrak{a}_k}$ decreases: In order for the target to pass over E_0 in one of the two cases, *all* previous isogenies have to succeed, for which Corollary 7 gives the probability $p_{\mathfrak{a}_k}$. Thus, the fact that SQALE first computes small-degree isogenies is slightly inconvenient for our attack, due to their low success probabilities. Nevertheless, attacking the last round of

SQALE-2048 has a success probability of roughly $p_{\mathfrak{a}_{221}} = \prod_{j=1}^{221}(\ell_j - 1)/\ell_j \approx 19.3\%$, so that in about 1 in 5 runs, every isogeny succeeds and we pass over E_0 for the 221-th bit, compared to 2 in 3 runs to pass over E_0 for the first bit ($p_{\mathfrak{a}_1} = \frac{2}{3}$). This means that we need about three times as many measurements to discover the last bit, than the first bit. Nonetheless the required total number of measurements for all bits is very managable; we get with Lemma 3:

Corollary 8. *Assuming a pass over E_0 leaks the k-th bit when the representation is strongly correlated, the estimated number of measurements to recover a SQALE-2048 key is*

$$4 \cdot \sum_{k=1}^{221} \frac{1}{p_{\mathfrak{a}_k}} = 4 \cdot \sum_{k=1}^{221} \prod_{i=1}^{k} \frac{\ell_i}{\ell_i - 1} \approx 4 \cdot 1020.$$

Here, the factor 4 represents the fact that we need to feed both $E_{\texttt{PK}}$ and $\tilde{E}_{\texttt{PK}}$, and that only half the time $(2C : 4C)$ is strongly-correlated. In practice, for more certainty, we increase the number of attempts per bit by some constant $\alpha > 1$, giving a total of $\alpha \cdot 4 \cdot 1020$ expected attempts. We detail this in Section 5.3.6.

5.3.3.3 Recovering secret keys in CTIDH

CTIDH [12] is the most efficient constant-time implementation of CSIDH to date, although the work restricts to the CSIDH-512 and CSIDH-1024 parameter sets. We note that techniques from CTIDH can be used to significantly speed up CSIDH for larger parameters too, yet this appears to require some modifications that have not been explored in the literature yet. In this section, we explain how zero-value curve attacks can be mounted on CTIDH, leading to a partial or full key recovery, depending on the number of measurements that is deemed possible. For concreteness, we focus on the CTIDH parameter set with a 220-bit key space, dubbed CTIDH-511 in [12], which uses 15 batches of up to 8 primes. The bounds satisfy $M_i \leq 12$.

Algorithmic description of CTIDH. Given a starting curve E_A, CTIDH computes the group action by multiple rounds of the following approach:

- Set $E \leftarrow E_A$, sample random points $P_+ \in E[\pi - 1]$ and $P_- \in E[\pi + 1]$.
- Per batch B_i, (attempt to) compute $\phi : E \to \mathfrak{l}_{i,j}^{\texttt{sign}(e_{i,j})} * E$ using P_+ resp. P_- (or dummy when all $\mathfrak{l}_{i,j}^{e_{i,j}}$ are performed). Push both points through each ϕ.
- Repeat this process until all $\mathfrak{l}_{i,j}^{e_{i,j}}$ and dummy isogenies have been applied.

Furthermore, the following design choices in CTIDH are especially relevant:

- Per batch B_i, CTIDH computes real isogenies first, and (potential) dummy isogenies after, to ensure M_i isogenies are computed, independently of $(e_{i,j})$.
- Per batch B_i, CTIDH computes the actual $\ell_{i,j}$-isogenies in ascending order.
- Per batch B_i, CTIDH scales the point rejection probability to the largest value, $1/\ell_{i,1}$. This slightly changes the computation of $p_{\mathfrak{a}_k}$.
- The order in which batches are processed is deterministic.

Example 1: Let $B_1 = \{3, 5\}$ with $M_1 = 6$, and let $e_{1,1} = 2$ and $e_{1,2} = -3$. For B_1, we first try tsectio compute $E \to \mathfrak{l}_1 * E$, until this succeeds twice. Then, we try to compute $E \to \mathfrak{l}_2^{-1} * E$, until this succeeds three times. After the real isogenies, we try to compute the remaining B_1-dummy isogeny. All B_1-isogenies, including dummies, have success probability $2/3$. If all six of the B_1-isogenies are performed but other B_i are unfinished, we skip B_1 in later rounds.

As for SQALE, the above description gives us that the order in which each $\mathfrak{l}$ is applied in CTIDH is deterministic, assuming that none of the steps fail, and so we get with Lemmas 2 and 3 again:

Corollary 9. *If no point rejections occur, the computation $\mathfrak{a} * E$ in CTIDH is an ordered evaluation $\mathfrak{l}^{(n)} * \ldots * \mathfrak{l}^{(1)} * E$, with $n = \sum M_i$, including dummy isogenies.*

Hence we can perform the adaptive attack on CTIDH-511 to recover the secret key bit by bit. The CTIDH implementation of [12] uses coefficients in alternative Montgomery form $(A + 2C : 4C)$, but passes over Montgomery form $(A : C)$ after each isogeny. Hence, E_0 will always have a zero-value representation and we detect E_0 as described in Section 5.3.3.1. In fact, we argue in Section 5.3.5 that zero-value representations are easier to detect than strongly-correlated representations.

Recovering the k-th bit. CTIDH introduces several difficulties for our attack, compared to SQALE. In particular, let $B_i = \{\ell_{i,1}, \ldots, \ell_{i,n_i}\}$ be the batch to be processed at step k. Then, since usually $n_i > 1$, we do not get a binary decision at each step as depicted in Figure 5.5, but a choice between $2n_i$ real isogeny steps $\mathfrak{l}_{i,j}^{\pm 1}$, or possibly a dummy isogeny. In practice, with high probability, we do not need to cover all $2n_i + 1$ options, as the following example shows.

Example 2: As CTIDH progresses through the batch ascendingly from $\ell_{i,1}$ to ℓ_{i,n_i}, the first step of a batch can often be recovered as in Figure 5.5, using public keys that are one $\ell_{i,1}$-isogeny away from E_0 respectively. If both do not pass over E_0, we deduce that $e_{i,1} = 0$, and we repeat this approach using an $\ell_{i,2}$-isogeny. In case of a successful attempt for $\ell_{i,j}$, we learn that the respective key element satisfies $e_{i,j} \leq -1$ resp. $e_{i,j} \geq 1$, depending on which of the binary steps was successful.[29] If we do not succeed in detecting E_0 after trying all $\ell_{i,j}^{\pm 1}$ in B_i, we learn that the target computes a B_i-dummy isogeny, and so all $e_{i,j} = 0$ for $\ell_{i,j} \in B_i$. We can easily confirm dummy isogenies: If the k-th step is a dummy isogeny, then using $E_{\mathtt{PK}}$ such that $\mathfrak{a} * E_{\mathtt{PK}}$ passes over E_0 in step $k-1$, we do not move to a different curve in step k and so we observe E_0 using side-channel information after steps $k-1$ *and* k.

Our approach to recover the k-th bit in CTIDH-511 only differs slightly from Section 5.3.3.2: Given the knowledge of the secret path up to step $k-1$, we recover the k-th step by iterating through the target batch $B_i = \{\ell_{i,1}, \dots, \ell_{i,n_i}\}$, until we detect E_0 for a given degree $\ell_{i,j}$, or otherwise assume a dummy isogeny. This iteration becomes easier in later rounds of each batch:

- If a previous round found that some $e_{i,j}$ is positive, we only have to check for positive $\ell_{i,j}$-isogeny steps later on (analogously for negative).
- If a previous round computed an $\ell_{i,j}$-isogeny, we immediately know that the current round cannot compute an $\ell_{i,h}$-isogeny with $h < j$.
- If a previous round detected a dummy isogeny for batch B_i, we can skip isogenies for B_i in all later rounds, since only dummy isogenies follow.

Thus, knowledge of the previous isogeny path significantly shrinks the search space for later steps. As in SQALE, the probability $p_{\mathfrak{a}_k}$ decreases the further we get: Batches containing small degrees ℓ_i appear multiple times, and steps with small ℓ_i have the most impact on $p_{\mathfrak{a}_k}$. For the last step $\mathfrak{l}^{(n)}$, the probability that *all* steps $\mathfrak{l}^{(k)}$ in CTIDH-511 succeed without a single point rejection, is roughly 0.3%. This might seem low at first, but the number of measurements required to make up for this probability does not explode; we are able to recover the full key with a reasonable amount of measurements as shown in Section 5.3.6. Furthermore, this probability represents the absolute lower bound, which is essentially the worst-case scenario: It is the probability that for the worst possible key, with no dummy isogenies, all steps must succeed in one run. In reality, almost all keys contain dummy isogenies, and we can relax the requirement that none of the steps fail, as failing dummy isogenies do not impact the curves passed afterwards.

Example 3: Let $B_1 = \{3, 5\}$ with $M_1 = 6$ as in CTIDH-511. Say we want to detect some step in the eighth round of some B_i for $i > 1$; it is not relevant in which of the seven former rounds the six B_1-isogenies are computed, and thus we can effectively allow for one point rejection in these rounds. This effect becomes more beneficial when dummy isogenies are involved. For example, if three of these six B_1-isogenies are dummies, we only need the three actual B_1-isogenies to be computed within the first seven rounds. Furthermore, after detecting the first dummy B_1-isogeny, we do not need to attack further B_1-isogenies as explained above, and therefore save significant attack effort.

Remark 12. *The generic attack requires that all first k steps succeed. This is not optimal: Assuming that some steps fail increases the probability of success of passing over E_0. For example, to attack isogenies in the sixth round and knowing that $\mathrm{e}_{1,1} = 5$, it is better to assume that one or two out of these five fail and will be performed after the $\ell_{i,j}$-isogeny we want to detect, than it is to assume that all five of these succeed in the first five rounds. This improves the success probability of passing over E_0 per measurement, but makes the analysis of the required number of measurements harder to carry out. Furthermore, this optimal approach highly depends on the respective private key. We therefore do not pursue this approach in our simulations. A concrete practical attack against a single private key that uses this improved strategy should require a smaller number of measurements.*

Remark 13. *For CTIDH with large parameters, one would expect more large ℓ_i and fewer isogenies of low degrees, relative to CTIDH-511. This improves the performance of our attack, as the probability of a full-torsion path increases, and so we expect more measurements to pass over E_0. However, the details of such an attack are highly dependent on the implementation of a large-parameter CTIDH scheme. As we know of no such implementation, we do not analyze such a hypothetical implementation in detail.*

Remark 14. *At a certain point, it might be useful to stop the attack, and compute the remaining key elements via a simple meet-in-the-middle search. Especially for later bits, if some dummy isogenies have been detected and most of the key elements $\mathrm{e}_{i,j}$ are already known, performing a brute-force attack may be faster than this side-channel attack.*

5.3.4 Recovering SIKE keys with side-channel leakage of E_6

We now apply the same strategy from Section 5.3.3 to SIKE. In this whole section, we focus on recovering Bob's key sk_B by showing side-channel leakage in $\mathtt{Derive}_B$, used in Decaps. In general, the idea would apply as well to recover Alice's key sk_A in static SIDH or SIKE with swapped roles, as we do not use any specific structure of 3-isogenies. One can easily verify that the attack generalizes to SIDH based on ℓ_A or ℓ_B-isogenies for *any* ℓ_A, ℓ_B. We repeat many of the general ideas from Section 5.3.3, with some small differences as SIKE operates in isogeny graphs over $\mathbb{F}_{p^2}$ instead of $\mathbb{F}_p$. Fortunately, these differences make the attack *easier*.

Detecting E_6. As remarked in Section 5.3.2, for both representations used in SIDH and SIKE, the curve E_6 is represented as $(8C : 4C)$, with $C = \alpha + \beta i \in \mathbb{F}_{p^2}$ non-zero. Similar to the CSIDH situation, whenever 4α or 4β is smaller than $p/2$, doubling $4C$ does not require a modular reduction for these values, and hence the bit representation of 8α resp. 8β of $8C$ is precisely a bit shift of 4α resp. 4β of $4C$ by one bit to the left. Such strongly-correlated values can be observed in several ways using side-channel analysis, as we detail later in Section 5.3.5. Different from the CSIDH situation are the following key observations:

- The prime used in SIKE is of the form $p = 2^{e_A} \cdot 3^{e_B} - 1$. As observed in Remark 9, this large cofactor 2^{e_A} in $p+1$ implies a modular reduction does *not* affect the lowest $e_A - 1$ bits, except for the shift. Hence, even when 4α or 4β is larger than $p/2$, we see strong correlation between their lowest bits.

- C is now an $\mathbb{F}_{p^2}$ value, so we get strong correlation between 8α and 4α *and* between 8β and 4β. This implies at least $2 \cdot (e_A - 1)$ strongly-correlated bits in the worst case (25 %), up to $2 \cdot (\log_2(p) - 1)$ strongly-correlated bits in the best case (25%).

For random curves E_a, the representations of a are indistinguishable from random $(\alpha + \beta i : \gamma + \delta i)$, and so an attacker can differentiate E_6 from such curves. From this, an attacker only needs a few traces to determine accurately whether a walk passes over E_6 or not, as discussed in Section 5.3.5.

General approach to recover the k-th trit. Assuming we know the first $k-1$ trits sk_i of a secret key sk, e.g. $\mathsf{sk}_{<k-1} = \sum_{i=0}^{k-2} \mathsf{sk}_i \cdot 3^i$, we want to find $\mathsf{sk}_{k-1} \in \{0,1,2\}$. We construct three secret keys, $\mathsf{sk}^{(0)}, \mathsf{sk}^{(1)}, \mathsf{sk}^{(2)}$ as

$$\mathsf{sk}^{(0)} = \mathsf{sk}_{<k-1} + 0 \cdot 3^{k-1}, \quad \mathsf{sk}^{(1)} = \mathsf{sk}_{<k-1} + 1 \cdot 3^{k-1}, \quad \mathsf{sk}^{(2)} = \mathsf{sk}_{<k-1} + 2 \cdot 3^{k-1}.$$

We must have $\mathsf{sk}_{<k} = \mathsf{sk}^{(i)}$ for some $i \in \{0,1,2\}$. Thus, we use these three keys to construct (see Lemma 4) three public keys $\mathsf{pk}^{(0)}, \mathsf{pk}^{(1)}, \mathsf{pk}^{(2)}$ such that $\texttt{Derive}_B(\mathsf{sk}^{(i)}, \mathsf{pk}^{(i)})$ computes E_6. When we feed these three keys to Bob, the computation $\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(i)})$ will then pass over E_6 in the k-th step *if and only if* $\mathsf{sk}_{k-1} = i$. By observing E_6 from side-channel information, we find sk_{k-1}.

In this attack scenario, another key observation makes the attack on SIDH and SIKE easier than the attack on CSIDH: The computation

$\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(i)})$ *always* passes over the same curves, as there are no "steps that can fail" as in CSIDH. We know *with certainty* that Bob will pass over E_6 in step k in precisely one of these three computations. Hence, the number of traces required reduces drastically, as we do not need to worry about probabilities, such as $p_{\mathfrak{a}_k}$, that we have for CSIDH.

Constructing $\mathsf{pk}^{(i)}$ from $\mathsf{sk}^{(i)}$ using backtracking. Whereas in CSIDH it is trivial to compute a curve E_{PK} such that $\mathfrak{a} * E_{\mathsf{PK}}$ passes over E_0 in the k-th step (see Lemma 2), in SIDH and SIKE it is not immediatly clear how to construct $\mathsf{pk}^{(i)}$ for $\mathsf{sk}^{(i)}$. We follow [2, § 3.3], using *backtracking* to construct such a pk.[30] The main idea is that any $\mathsf{sk}_{<k}$ corresponds to some *kernel point* R_B of order 3^k for some k, so to an *isogeny* $\phi^{(k)} : E_6 \to E^{(k)}$. Here, the trits sk_i determine the steps

$$E_6 = E^{(0)} \xrightarrow{\mathsf{sk}_0} E^{(1)} \xrightarrow{\mathsf{sk}_1} \dots \xrightarrow{\mathsf{sk}_{k-1}} E^{(k)}.$$

The dual isogeny $\hat{\phi}^{(k)} : E^{(k)} \to E_6$ then corresponds to the kernel generator $\phi^{(k)}([3^{e_B-k}]Q_B)$ (see [145]). This leads to [2, Lemma 2].

Lemma 4. *Let* sk *be a secret key, and let* $R_k = [3^{e_B-k}](P_B + [\mathsf{sk}_{<k}]Q_B)$ *so that* $\phi : E_6 \to E^{(k)}$ *is the corresponding isogeny for the first k steps. Let* $T \in E^{(k)}[3^{e_B}]$ *such that* $[3^{e_B-k}]T \neq \pm[3^{e_B-k}]\phi(Q_B)$*. Then*

$$\mathsf{pk}' = (\phi(Q_B) + [\mathsf{sk}_{<k}]T,\ -T,\ \phi(Q_B) + [\mathsf{sk}_{<k} - 1]T)$$

is such that $\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}')$ *passes over* E_6 *in the k-th step.*

It is necessary that such a pk' is not rejected by a SIKE implementation.

Corollary 10 ([2])**.** *The points* P' *and* Q' *for a* $\mathsf{pk}' = (P', Q', Q' - P')$ *as constructed in Lemma 4 form a basis for the* 3^{e_B}*-torsion of* $E^{(k)}$*. This implies they are of order* 3^{e_B} *and pass the CLN test.*

Given Lemma 4 and $\mathsf{sk}_{<k-1}$, we can therefore easily compute the $\mathsf{pk}^{(i)}$ corresponding to $\mathsf{sk}^{(i)}$ for $i \in \{0, 1, 2\}$. One of the three attempts $\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(i)})$ will then pass over E_6 in the k-th step, while the other two will not. Only the representation of E_6 by $(8C : 4C)$ is then strongly-correlated, and by detecting this representation using side-channel information, we recover sk_{k-1}.

Remark 15. *A straightforward attack computes* $\mathsf{pk}^{(0)}, \mathsf{pk}^{(1)}$ *and* $\mathsf{pk}^{(2)}$*, and feeds all three to Bob, and so requires 3 traces to recover a single trit* sk_{k-1}*. Clearly, when we already detect* E_6 *in the trace of* $\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(0)})$*, we do not need the traces of* $\mathsf{pk}^{(1)}$ *and* $\mathsf{pk}^{(2)}$*, similarly for* $\texttt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(1)})$.

[30] It is important that such a $\mathsf{pk} = (P, Q, Q - P)$ passes the *CLN test* [64]: P and Q are both of order 3^{e_B} and $[3^{e_B-1}]P \neq [\pm 3^{e_B-1}]Q$, so that they generate $E[3^{e_B}]$.

This approach would require on average $\frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 3 = 2$ *traces per trit. We can do even better: if we do not detect* E_6 *in both* $\mathtt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(0)})$ *and* $\mathtt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(1)})$*, we do not need a sample for* $\mathtt{Derive}_B(\mathsf{sk}, \mathsf{pk}^{(2)})$*, as* sk_{k-1} must *equal 2. This gives* $\frac{5}{3}$ *samples per trit, giving a total of* $\frac{5}{3} \cdot e_B$ *traces.*

5.3.5 Feasibility of obtaining the side-channel information

In this section, we discuss the practical feasibility of obtaining the required side-channel information.

Zero-value representations. For zero-value representations as in CTIDH, where E_0 is represented by $(0 : C)$ in Montgomery form, we exploit side-channel analysis methods to distinguish between the zero curve and others. In particular, as shown in [74], one can apply Welch's t-test [165] to extract the required information from the power consumption of the attacked device. Further, as mentioned in [74], one can use correlation-collision SCA methods to identify zero values using multiple measurements. Therefore, the attack scheme as demonstrated in [74] to SIKE can analogously be applied whenever zero-value representations occur.

Strongly-correlated representations. Our attack strategy for implementations using strongly-correlated representations, such as SQALE and SIKE, is more challenging in practice, since no zero values occur. A naïve approach to mount our attack for such instances would be to apply side-channel attacks like CPA or DPA, and estimate or guess the values of intermediate codomain curves. Revealing those intermediate values would require a fitting power model and a sufficiently high signal-to-noise ratio (SNR).[31] By exploiting the pattern similarity in the strongly-correlated representation $(2C : 4C)$ for SQALE or $(8C : 4C)$ for SIKE, as mentioned in Section 5.3.3.1 and Section 5.3.4, we reduce the SNR required for successfully performing the attack. To achieve this, we apply the concept of correlation-collision attacks, so that there is no need to reveal the actual value of C via a sophisticated power model.

We exploit side-channel correlation-collision attacks [142] to find similar values by searching for strongly-correlated patterns versus non-correlated patterns. Instead of measuring *multiple* computations to identify similar or identical patterns, as in [142], we apply the concept of a horizontal side-channel attack as in [147]. That is, we extract the required side-channel information from a *single* segmented power trace. Such a segmented power

[31]SNR is the ratio between the variance of the signal containing the desired information and the variance of noise. Too small SNR values make information and noise indistinguishable.

trace contains the power values of the processed limbs (each limb is 64 bits), required to represent $\mathbb{F}_p$-values, which form a fingerprint characteristic of such a value. These fingerprints then serve as input to calculate the correlation between $2C$ and $4C$ for SQALE, or $4\alpha, 4\beta, 8\alpha$ and 8β for SIKE, from which we judge their similarity. For strongly-correlated representations of E_0 and E_6, this gives a higher correlation between the fingerprints than for representations of random curves E_a as either $(A+2C : 4C)$ or $(A+2C : A-2C)$, with $A, C \neq 0$.

For both CSIDH attacks, we assume no point rejections prior to the respective isogeny computation, so that the specific isogeny steps are known in advance. For SIKE, there are no such probabilities involved in the isogeny computation, and so here too the specific isogeny steps are known in advance. Thus, in all cases, the points of interest (position of the limbs) within the power trace are known in advance, and segmenting each power trace into vectors of the corresponding processed limbs for mounting the correlation-collision attack is easy.

5.3.6 Simulating the attacks on SQALE, CTIDH and SIKE

To demonstrate our attacks, we implemented Python (version 3.8.10) simulations for our CTIDH-511[32] and SQALE-2048[33] attacks, and a C simulation of the attack on SIKE.[34] The C code for key generation and collecting the simulated power consumption were compiled with gcc (version 9.4.0). Security-critical spots of the attacked C code remained unchanged in both cases.

For the SQALE and CTIDH attacks, the implemented simulation works as follows: First, we generate the corresponding public keys E_{PK} and $\tilde{E}_{\text{PK}}$ for the current k-th step, as described in Section 5.3.3.1. Then we collect the bit values of the resulting codomain curve after the computation of the k-th step $E \leftarrow \mathfrak{l}^{(k)} * E$ in the group action $\mathfrak{a} * E_{\text{PK}}$ resp. $\mathfrak{a} * \tilde{E}_{\text{PK}}$ to simulate the power consumption.

We calculate the Hamming weight of these values and add a zero-mean Gaussian standard distribution to simulate noise in the measurement. We picked different values of the standard deviation to mimic realistic power measurements with different SNR values. By varying the SNR in such a way, we can determine up to which SNR our attacks are successful, and compare this to known SNR values achieved in physical attacks. For SQALE and CTIDH, we are only interested in power traces passing over E_0, and so we need the first k steps to succeed. We therefore take enough samples to ensure high probability that passing over E_0 happens multiple times for either E_{PK}

[32]`http://ctidh.isogeny.org/high-ctidh-20210523.tar.gz`
[33]`https://github.com/JJChiDguez/sqale-csidh-velusqrt`, commit `a95812f`
[34]`https://github.com/Microsoft/PQCrypto-SIDH`, commit `ecf93e9`

or $\tilde{E_{\mathsf{PK}}}$. Finally, based on the set of collected bit vectors for all these samples, we decide on which of the two cases contains paths over E_0, and therefore reveal the k-th bit of the secret key.

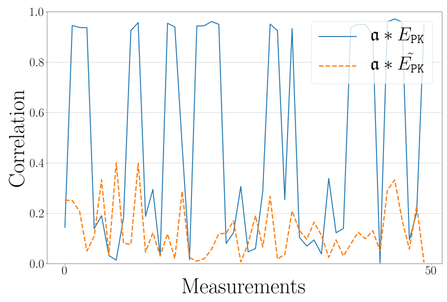
For the SIKE attack, we generate $\mathsf{pk}^{(0)}, \mathsf{pk}^{(1)}$ and $\mathsf{pk}^{(2)}$ for the current k-th step, as described in Section 5.3.4, and collect the bit values of the resulting codomain curve in the computation of the k-th step of $\mathtt{Derive}_B$ in Decaps. For simplicity, there is no noise in the simulation, as the results are exactly the same as for the SQALE situation after extracting the bit values. Deciding which sample has strongly-correlated values is easy, as is clear from Figure 5.7.

As described in previous sections, due to the different representations, the decision step differs between CTIDH and SQALE. For SIKE, the probability to pass over E_6 is 100%, and so a single sample per $\mathsf{pk}^{(i)}$ is enough to decide what the k-th trit sk_{k-1} is.

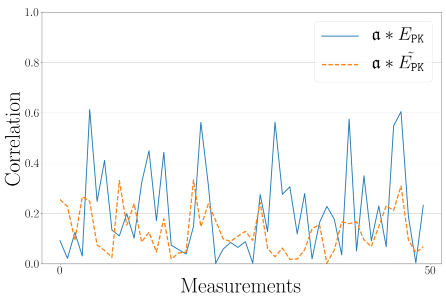
In order to reduce the running time of our simulations for SQALE and CTIDH, we terminate each group action run after returning the required bit values of the k-th step. Furthermore, we implemented a threaded version so that we collect several runs in parallel, which speeds up the simulation. All experiments were measured on AMD EPYC 7643 CPU cores.

Attacking CTIDH-511. As shown in [74, § 4] a practical differentiation between zero and non-zero values, even with low SNR, is feasible with a single trace containing the zero value. Hence, in CTIDH, where E_0 is represented by $(0 : C)$, a single occurrence of E_0 leaks enough information for the decision in each step. Thus, the number of required attempts can be calculated as follows: Given $p_{\mathfrak{a}_k}$ from Lemma 3, the success probability of having *at least one* sequence that passes over E_0 in the k-th step in t_k attempts is $P_{exp}(X \geq 1) = 1 - (1 - p_{\mathfrak{a}_k})^{t_k}$. We can calculate t_k to achieve an expected success rate P_{exp} by $t_k = \log_{(1-p_{\mathfrak{a}_k})}(1 - P_{exp})$. For CTIDH-511, to achieve $P_{exp} \geq 99\%$ for all k, we get an estimate of $\sum t_k \approx 130,000$ attempts for full key recovery. In simulations, the required number of attempts for full key recovery was $\approx 85,000$ on average over 100 experiments, due to effects mentioned in Section 5.3.3.3. The average execution time was about 35 minutes (single core) or 5 minutes (120 threads). We note that finding key bits of the last round(s) via meet-in-the-middle searches will often be feasible in practice, which reduces the number of required measurements significantly due to the small values of $p_{\mathfrak{a}_k}$ for large k.

Attacking SQALE-2048. In this case, we simulate a correlation-collision attack as described in Section 5.3.5: We calculate the correlation between the 64-bit limbs that represent the $\mathbb{F}_p$-values, and apply the standard Hamming-weight model with noise drawn from a normal distribution. Even with an SNR as low as 1.40, strongly-correlated representations leak enough information to guess the k-th bit, as can be seen in Figure 5.7 for $k = 1$. Both



(a) Correlation results without noise.



(b) Correlation results with SNR of 1.40.

Figure 5.7: Experimental results to discover bit $k = 1$: the correct hypothesis $(\mathfrak{a} * E_{\text{PK}})$ in blue and the wrong hypothesis $(\mathfrak{a} * \tilde{E_{\text{PK}}})$ in orange, for SQALE-2048.

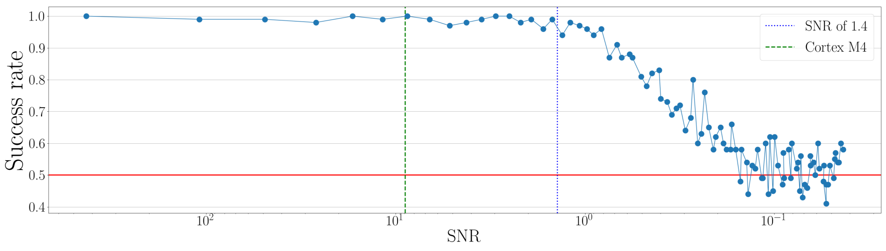


Figure 5.8: Relation between SNR and success rate. Rate of 0.5 equals random guess.

without noise and with SNR 1.40, we are able to determine the right bit in 74% of measurements (where 75% is the theoretical optimum, as $2C \leq \frac{p}{2}$ only half the time). An SNR value of 1.40 is considered *low*: The SNR value of a common embedded device, using a measurement script[35] provided by the ChipWhisperer framework for a ChipWhisperer-Lite board with an ARM Cortex-M4 target, obtains an SNR of 8.90. Figure 5.8 shows the success rate for different values. We evaluated the following methods for decision-making:

- Decide based on the number of cases with a higher resulting correlation, as exemplified in Figure 5.7.
- Decide based on the sum of the resulting correlations for each case, to reduce the number of attempts required for a given success rate.

[35]`https://github.com/newaetech/chipwhisperer-jupyter/blob/master/archive/PA_Intro_3-Measuring_SNR_of_Target.ipynb`, commit 44112f6

Empirical results show that the sum-based approach reduces the required number of attempts for full key recovery by a factor of 3 on average (from $\approx 24,819$ to $\approx 8,273$), which leads to an average execution time of ≈ 35 minutes (120 threads).

Attacking SIKE. For SIKE too, we simulate a correlation-collision attack as described in Section 5.3.5: We calculate the correlation between the 64-bit limbs that represent the $\mathbb{F}_p$-values, and apply the standard Hamming-weight model with noise drawn from a normal distribution. The analysis is entirely similar to that of the SQALE case, with the exceptions that **i)** we know that one of the three samples per trit must be an E_6-sample, **ii)** we know that even with modular reduction, there is strong correlation between the lowest limbs and **iii)** we can use both $\mathbb{F}_p$-values α and β for $C = \alpha + \beta i \in \mathbb{F}_{p^2}$.

As explained in Section 5.3.4, we need on average 5/3 samples per trit to find sk_i, for all e_B trits. For SIKEp434, this gives an average of 228 samples to recover sk_B. The average running time over 100 evaluations in each case was ≈ 4 seconds for SIKEp434, ≈ 8 seconds for SIKEp503, ≈ 17 seconds for SIKEp610, and ≈ 42 seconds for SIKEp751 respectively.

Scheme	SQALE-2048	CTIDH-511	SIKEp434	SIKEp503	SIKEp610	SIKEp751
Samples	8,273	85,000	228	265	320	398

Table 5.7: Required number of samples to reconstruct secret key in simulations.

5.3.7 Countermeasures and conclusion

We have shown that both Montgomery form and alternative Montgomery form leak secret information when passing over E_0. As described in Section 5.3.5, zero-value representations are easiest to detect, and accordingly one should prefer the alternative Montgomery form over the Montgomery form throughout the whole computation. However, more effective countermeasures are required to avoid strongly-correlated representations.

Avoiding E_0. A straightforward way of mitigating the attacks is to avoid paths that lead over E_0. Intuitively, randomizing the order of isogenies, and as proposed in [124] the order of real and dummy isogenies, might seem beneficial to achieve this. However, we can then simply *always* attack the first step of the isogeny path, with a success probability of $1/n$. With enough repetitions, we can therefore statistically guess the secret key, where the exact success probabilities highly depend on the respective CSIDH variant. Thus, this countermeasure merely leads to a moderately higher number of required measurements, but still allows for full key recovery, e.g. in SQALE.

Furthermore, randomizing the order in which isogenies are evaluated has a significant impact on the performance, since all state-of-the-art constant-time implementations of CSIDH use specific strategies for efficiency that fix this order accordingly.

Similarly, it may appear intuitive to define a certain *danger zone* around E_0, e.g., containing all curves $\mathfrak{l}_i^{\pm 1} * E_0$ for $1 \leq i \leq n$, and abort the execution of the protocol whenever an isogeny path enters this zone. However, the attacker can simply construct public keys that would or would not pass through this zone, and observe that the protocol aborts or proceeds. This leaks the same information as in the attack targeting only E_0.[36]

One can fully bypass this danger zone by masking by a (small) isogeny $\mathfrak{z}$ before applying any secret $\mathfrak{a}$. By commutativity, $\mathfrak{a} * E = \mathfrak{z}^{-1} * (\mathfrak{a} * (\mathfrak{z} * E))$, so this route avoids the danger zone when $\mathfrak{z}$ is sufficiently large. Drawing $\mathfrak{z}$ from a masking key space of k bits would require the attacker to guess the random ephemeral mask correctly in order to get a successful walk over E_0, which happens with probability 2^{-k}. Thus, a k-bit mask increases the number of samples needed by 2^k. Similar countermeasures have been proposed in [120, 2]. Although masking comes at a significant cost if $\mathfrak{z}$ needs to be large, this appears to be the only known effective countermeasure that fully avoids our attacks.

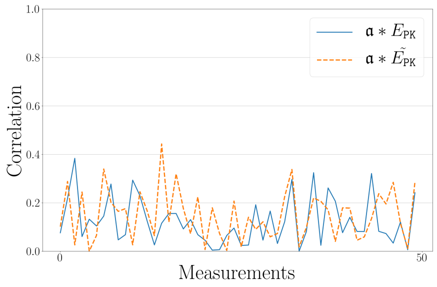
Avoiding correlations in alternative Montgomery form. As noted, $(2C : 4C)$ leaks secret information whenever $2C < \frac{p}{2}$. In order to avoid this, we can try to represent the alternative Montgomery form $(A + 2C : 4C)$ differently and use a *flipped* alternative Montgomery form $(A + 2C : -4C)$ instead, which we write as ǝʌᴉʇɐuɹǝʇlɐ Montgomery form for brevity. In the case of E_0, this means that the coefficients $2C$ and $-4C$ are *not* simple shifts of each other for $2C < \frac{p}{2}$, which prevents the correlation attack. In order to still achieve constant-time behavior, we should flip $4C$ for all curves, since otherwise E_0 would easily be detectable via side channels. The correctness of computations can be guaranteed by corresponding sign flips in computations that would normally include $4C$.

Although the ǝʌᴉʇɐuɹǝʇlɐ Montgomery form is effective in preventing leakage of E_0, it creates other vulnerable curves. We discuss this in more detail in Section 5.3.8. It remains an open question to find a representation without both zero-value representations and strongly-correlated representations.

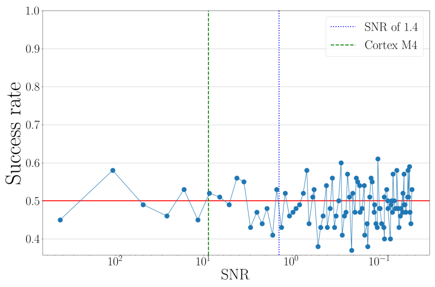
[36]Pun afficionados may wish to dub this scenario the *highway to the danger zone.*

5.3.8 Flipping $4C$ as a countermeasure.

In this appendix, we discuss the effectiveness of ǝʌᴉʇɐuɹǝʇlɐ Montgomery form as a countermeasure. As Figure 5.9 shows, the countermeasure prevents detection of $(2C : 4C)$, and therefore prevents leakage on E_0. Similar techniques can also be applied for other strongly-correlated representations, such as those for E_6.



(a) Correlation results with countermeasures (without noise).



(b) SNR/Success rate with countermeasures.

Figure 5.9: Correlation values including the countermeasures leak no information.

Nevertheless, ǝʌᴉʇɐuɹǝʇlɐ Montgomery form creates new problems. In alternative Montgomery form, the curve E_{-6}, a valid supersingular curve for most CSIDH-primes, is represented using $(-4C : 4C)$, which is not strongly-correlated. However, by switching to ǝʌᴉʇɐuɹǝʇlɐ Montgomery form, we get the strongly-correlated representation $(-4C : -4C)$. The attack as described in the paper is then applicable by replacing E_0 by E_{-6}.

It seems difficult to discover if a curve E_a will leak information before computing the next step: doing so would require knowledge on a, and so requires a representation of a in some form. Hence, we cannot decide to use either alternative Montgomery form or ǝʌᴉʇɐuɹǝʇlɐ Montgomery form before we compute the actual curve.

5.3.9 CSIDH implementations using radical isogenies

In [47] an alternative method to compute the action of $\mathfrak{a}$ is proposed, using *radical isogenies*. The evaluation of $\mathfrak{a} * E$ is still an ordered evaluation $\mathfrak{l}^{(n)} * \ldots * \mathfrak{l}^{(1)} * E$, due to a change in the evaluation algorithm we get chains $\mathfrak{l}_i * \ldots * \mathfrak{l}_i$ of specific degrees $\ell_i \in \{4, 5, 7, 9, 11, 13\}$ in the evaluation. These chains are computed on a different curve form, namely the Tate normal form for that specific degree ℓ_i, instead of as steps between Montgomery curves.

$$\begin{tikzcd}
\cdots \arrow[r, "\mathfrak{l}^{(j)}"] & E \arrow[d, "\text{To Tate normal form}"] & & & \mathfrak{l}_i^k * E \arrow[r, "\mathfrak{l}^{(j+k)}"] & \cdots \\
& E(b_0, c_0) \arrow[r, "\text{Rad.}"] & E(b_1, c_1) \arrow[r, "\text{Rad.}"] & \cdots \arrow[r, "\text{Rad.}"] & E(b_k, c_k) \arrow[u, "\text{To Montgomery}"] &
\end{tikzcd}$$

The Tate normal form for a degree ℓ_i in general requires two coefficients $b, c \in \mathbb{F}_p$ instead of the single Montgomery coefficient $a \in \mathbb{F}_p$, and the radical isogeny computes b', c' associated to $\mathfrak{l}_i * E_a$.

In an efficient implementation, both b and c would be represented in projective coordinates. We know of only one such implementation, given in [55]. We sketch two attack approaches to extend our attack to such an implementation:

1. Find a Tate normal curve of degree ℓ_i such that either b or c has a strongly-correlated representation. The generic adaptive attack then works exactly the same.

2. Find the length of the chain by feeding a curve $E_{\mathtt{PK}}$ such that we map back to E_0 when we map back to Montgomery form at the end of the chain. This requires feeding several different $E_{\mathtt{PK}_j}$ representing several different lengths of chains.

Note that the attack becomes easier when using radical isogenies: these chains are computationally very distinct from ordinary isogeny evaluations, and so we only need to discover the length of the chain. Furthermore, radical isogenies are performed for low degrees (up to 13), which implies that we do not perform these degrees in the rest of the steps $\mathfrak{l}^{(j)}$. This increases $p_{\mathfrak{a}_k}$ substantially.

6
Applications

6.1 On Actively Secure Fine-grained Access Structures from Isogeny Assumptions

This chapter is for all practical purposes identical to the paper *On Actively Secure Fine-grained Access Structures from Isogeny Assumptions* [45] authored jointly with Philipp Muth, which was published at PQCrypto 2022.

6.1.1 Introduction

The principal motivation for a secret sharing scheme is to split private information into fragments and securely distribute these shares among a set of shareholders. Then, any collaborating set with a sufficient number of participants is able to reconstruct the shared private information, while the secret remains confidential to any unauthorised, that is not sufficiently large, subset of shareholders.

Since their introduction in the 1970s by Blakley [31] and Shamir [168], the field of secret sharing schemes, information theoretic and computational, has been studied extensively. In previous years, due to applications in blockchain and other scenarios, the interest in new developments and applications for secret sharing schemes has increased.

Post-quantum schemes have, however, only received little attention with respect to secret sharing. Recently, De Feo and Meyer [73] proposed a key exchange mechanism and a signature scheme making use of isogeny based public key cryptography for which the secret key is stored in a Shamir shared way. Their approach enables decapsulation for the key exchange mechanism and signing for the signature scheme in a round-robin way without reconstructing the secret key in clear for any sufficiently large set of shareholders. Yet in applying Shamir's secret sharing scheme they restrict themselves to simple threshold access structures. Furthermore, their protocols are only passively secure, in that while a misbehaving shareholder cannot obtain information on the secret key shares of other shareholders participating in a decapsulation or a signing execution via maliciously formed inputs, his deviation from the protocol cannot be detected.

We aim to tackle both caveats by proposing an actively secure isogeny based key exchange mechanism, for which the secret key is secret shared by a trusted dealer. We further transform the key exchange mechanism into an actively secure signature scheme with shared secret key.

Our Contribution. Our contribution is manifold. First, we transfer the active security measures outlined in [27] from their setting of full engagement protocols to a setting of threshold secret sharing. We thereby open the active security measures to a wider field of application and improve upon their efficiency significantly. Second, we apply the adapted active security measures to propose an actively secure key exchange mechanism with secret shared secret key. Third, we present an actively secure signature scheme by applying a Fiat-Shamir transform to our key exchange mechanism. And fourth, we expand our key encapsulation mechanism and our signature scheme to a wider field of secret sharing schemes. For that we characterise the necessary properties for a secret sharing scheme and give several examples of compatible schemes.

Related work. Secret sharing schemes were first introduced by Blakley [31] and Shamir [168]. In both their approaches, secrets from the secret space $\mathbb{Z}_p := \mathbb{Z} \bmod p$ for prime p are shared by distributing interpolation points of randomly sampled polynomials. Damgård and Thorbek [69] presented a secret sharing scheme with secret space $\mathbb{Z}$. Thorbek [179] later improved their scheme. Yet their scheme is only computationally confidential, compared to the information theoretical confidentiality of Shamir and Blakley's schemes. Tassa [176] opened Shamir's scheme to a more general application by utilising the derivatives of the sharing polynomial to construct a hierarchical access structure. These basic secret sharing schemes rely on the dealer providing honestly generated shares to the shareholders. Verifiable secret sharing schemes eliminate this drawback by providing the shareholders with the means to verify the correctness of the received shares with varying overhead. Examples of these are [25, 157, 174]. With minor efficiency losses, Herranz and Sáez [102] were able to achieve verifiable secret sharing for generalised access structures. Traverso, Demirel, and Buchmann [181] proposed an approach for evaluating arithmetic circuits on secret shared in Tassa's scheme, that also enabled auditing the results. Cozzo and Smart [67] investigated the possibility of constructing shared secret schemes based on the Round 2 candidate signature schemes in the NIST standardization process[37]. Based on CSI-FiSh [28], De Feo and Meyer [73] introduced threshold variants of passively secure encryption

[37]`https://csrc.nist.gov/Projects/post-quantum-cryptography/Post-Quantum-Cryptography-Standardization`

and signature schemes in the Hard Homogeneous Spaces (HHS) setting. Cozzo and Smart [68] presented the first actively secure but not robust distributed signature scheme based on isogeny assumptions. In [27], the authors presented CSI-RAShi, a robust and actively secure distributed key generation protocol based on Shamir's secret sharing in the setting of HHS, which necessitates all shareholders to participate.

Outline. In Section 6.1.2 the terminology, primitives and security notions relevant for this work are introduced. Section 6.1.3 presents an actively secure threshold key exchange mechanism and proves our scheme's active security and simulatability. The actively secure signature scheme resulting from applying the Fiat-Shamir-transform to our key exchange mechanism is discussed in Section 6.1.4. Finally, the necessary properties for a secret sharing scheme to be compatible with our key exchange mechanism and signature scheme are characterised in Section 6.1.5 in order to enable applying a more general class of secret sharing schemes.

6.1.2 Preliminaries

Notation. Throughout this work we use a security parameter $\lambda \in \mathbb{N}$. It is implicitly handed to a protocol whenever needed, that is protocols with computational security. Information theoretic schemes and protocols such as secret sharing schemes used in this work do not require a security parameter.

For an indexed set $X = \{x_i\}_{i \in I}$, we denote the projection onto a subset $I' \subset I$ by $X_{I'} = \{x_i \in X : i \in I'\}$. The same holds for indexed tuples $(x_i)_{i \in I}$.

Secret Sharing Schemes. A secret sharing scheme is a cryptographic primitive that allows a dealer to share a secret among a set of shareholders. An instance is thus defined by a secret space G, a set of shareholders S and an access structure $\Gamma_{\mathcal{S}}$. A set $S' \in \Gamma_{\mathcal{S}}$ is called *authorised* and can from their respective shares reconstruct a shared secret. If the instance $\mathcal{S}$ is clear from the context, we omit the index in the access structure Γ. In this work, we consider monotone access structures, that is for any $A \subset B \subset S$ with $A \in \Gamma$, we also have $B \in \Gamma$.

A secret sharing instance $\mathcal{S}$ provides two algorithms: Share and Rec. A dealer executes $\mathcal{S}.\mathsf{Share}(s)$ to generate shares $s_1, \ldots, s_k$ of a secret s. A share s_i is assigned to a shareholder $P_{\phi(i)}$ via a surjective map $\phi : \{1, \ldots, k\} \to \{1, \ldots, n\}$ induced by $\Gamma_{\mathcal{S}}$. A set of shareholders $S' \in \Gamma_{\mathcal{S}}$ executes

$$\mathcal{S}.\mathsf{Rec}\Big(\{s_i\}_{P_{\phi(i)} \in S'}\Big)$$

on their respective shares to retrieve a previously shared secret.

Definition 24 (Superauthorised sets). *For a secret sharing instance $\mathcal{S} = (G, S, \Gamma_{\mathcal{S}})$, we call a set $S' \subset S$* superauthorised, *if for any $P \in S'$, we have*

$S' \setminus \{P\} \subseteq \Gamma_{\mathcal{S}}$. We denote the set of superauthorised sets of shareholders by $\Gamma^+_{\mathcal{S}}$.

Any superauthorised set is also authorised.

Example 1:[Shamir's secret sharing] An instance of Shamir's famous secret sharing scheme consists of a set of $n > 0$ shareholders, a secret space $\mathbb{Z} \bmod p$, where p is a prime larger than n, and an access structure $\Gamma = \{S' \subset S : \#S' \geq t\}$ for a threshold $t \leq n$. A secret $s \in \mathbb{Z} \bmod p$ is shared by handing each shareholder P_i an interpolation point of a randomly sampled polynomial of degree $t-1$ with constant term s. Reconstruction is achieved via Lagrange interpolation, that is

$$s = \sum_{P_i \in S'} L_{i,S'} s_i = \sum_{P_i \in S'} \prod_{\substack{P_j \in S' \\ j \neq i}} \frac{j}{j-i} f(i)$$

for some $S' \in \Gamma$ and Lagrange interpolation coefficients $L_{i,S'}$.

Hard Homogeneous Spaces. We present our key exchange mechanism and signature scheme in the context of *hard homogeneous spaces* (HHS). HHS were first discussed by Couveignes [66] in 2006. He defines a HHS $(\mathcal{E}, \mathcal{G})$ as a set $\mathcal{E}$ and a group $(\mathcal{G}, \odot)$ equipped with a transitive action $* : \mathcal{G} \times \mathcal{E} \to \mathcal{E}$. This action has the following properties:

- Compatibility: For any $g, g' \in \mathcal{G}$ and any $E \in \mathcal{E}$, we have $g * (g' * E) = (g \odot g') * E$.
- Identity: For any $E \in \mathcal{E}$, $i * E = E$ if and only if $i \in \mathcal{G}$ is the identity element.
- Transitivity: For any $E, E' \in \mathcal{E}$, there exists exactly one $g \in \mathcal{G}$ such that $g * E = E'$.

Definition 25 (Notation)**.** *For a HHS $(\mathcal{E}, \mathcal{G})$ with a fixed $g \in \mathcal{G}$, let $p | \#\mathcal{G}$ be a fixed prime. We denote $[s]E := g^s * E$ for all $s \in \mathbb{Z}_p$ and all $E \in \mathcal{E}$.*

The following problems are assumed to be efficiently computable in a HHS $(\mathcal{E}, \mathcal{G})$, i.e., there exist polynomial time algorithms to solve them:

- Group operations on $\mathcal{G}$ (membership, inverting elements, evaluating $\odot$).
- Sampling elements of $\mathcal{E}$ and $\mathcal{G}$.
- Testing the membership of $\mathcal{E}$.

- Computing the transitive action $\star$: given $g \in \mathcal{G}$ and $E \in \mathcal{E}$ as input, compute $g \star E$.

Whereas the subsequent problems are assumed to be hard in a HHS $(\mathcal{E}, \mathcal{G})$.

Problem 9 (Group Action Inverse Problem (GAIP)). *Given two elements $E, E' \in \mathcal{E}$ as input, the challenge is to provide $g \in \mathcal{G}$ with $E' = g \star E$. Due to the transitivity property of HHS a given instance of the GAIP has a solution.*

Problem 10 (Parallelisation Problem). *An instance of the* Parallelisation Problem *is defined by a triple $(E, E', F) \in \mathcal{E}^3$ with $E' = g \star E$. The challenge is to provide F' with $F' = g \star F$.*

The intuitive decisional continuation of this problem is as follows.

Problem 11 (Decisional Parallelisation Problem). *An instance of the* Decisional Parallelisation Problem *is defined by a base element $E \in \mathcal{E}$ and a triple (E_a, E_b, E_c) with $E_a = [a]E$, $E_b = [b]E$ and $E_c = [c]E$. The challenge is to distinguish whether $c = a + b$ or $c \leftarrow\$ \mathbb{Z}_p$ was randomly sampled.*

Remark 16. *It is obvious that the decisional parallelisation problem reduces to the parallelisation problem, which reduces to the group action inverse problem.*

Threshold Group Action. Let s be a Shamir shared secret among shareholders $P_1, \dots, P_n$, that is each P_i holds a share s_i of s, $i = 1, \dots, n$. To compute $E' = [s]E$ for an arbitrary but fixed $E \in \mathcal{E}$ without reconstructing s, we have an authorised set of shareholders execute Algorithm 24. If it is executed successfully, we have by the compatibility property of $\star$ and the repeated application of $E^k \leftarrow [L_{i,S'} s_i] E^{k-1}$ the result

$$E^{\#S'} = \left[\sum_{P_i \in S'} L_{i,S'} s_i \right] E = [s] E.$$

Piecewise Verifiable Proofs. A piecewise verifiable proof (PVP) is a cryptographic primitive in the context of hard homogeneous spaces and was first introduced in [27]. It is a compact non-interactive zero-knowledge proof of knowledge of a witness $f \in \mathbb{Z}_q[X]$ for a statement

$$x = ((E_0, E_1), s_1, \dots, s_n), \tag{6.1}$$

with statement pieces $s_i = f(i)$ for $i = 0, \dots, n$, with $E_1 = [s_0] E_0 \in \mathcal{E}$. A PVP provides a proving protocol $\mathsf{PVP}.P$, which takes a statement x of the form (6.1) and a witness f and outputs a proof $\left(\pi, \{\pi_i\}_{i=0,\dots,n}\right)$, where (π, π_i) is a proof piece for s_i, $i = 0, \dots, n$. The PVP also provides a verifying protocol $\mathsf{PVP}.V$, which takes an index $i \in \{0, \dots, n\}$, a statement piece s_i and a proof piece (π, π_i) and outputs true or false. Let $\mathcal{R} = \{(x, f)\}$, where f is a witness for the statement x. The projection R_I for some $I \subset \{0, \dots, n\}$ denotes (x_I, f).

Definition 26 (Completeness)**.** *We call a PVP* complete*, if, for any* $(x, f) \in \mathcal{R}$ *and*

$$\left(\pi, \{\pi_i\}_{i=0,\ldots,n}\right) \leftarrow \mathsf{PVP}.P\,(f, x)\,,$$

the verification succeeds, that is

$$\forall j \in \{0, \ldots, n\} \colon \Pr[\mathsf{PVP}.V\,(j, x_j, (\pi, \pi_j)) = \mathsf{true}] = 1.$$

Definition 27 (Soundness)**.** *A PVP is called* sound *if, for any adversary $\mathcal{A}$, any $I \subset \{0, \ldots, n\}$ and any x for which there exists no f with $(x_I, f) \in \mathcal{R}_I$,*

$$\Pr[\mathsf{PVP}.V(j, x_j, (\pi, \pi_j)) = \mathsf{true}]$$

is negligible in the security parameter λ for all $j \in I$, where $(\pi, \{\pi_i\}_{i \in I}) \leftarrow \mathcal{A}(1^\lambda)$.

Definition 28 (Zero-knowledge)**.** *A PVP is* zero-knowledge*, if for any $I \subset \{1, \ldots, n\}$ and any $(x, f) \in \mathcal{R}$, there exists a simulator* Sim *such that for any polynomial-time distinguisher $\mathcal{A}$ the advantage*

$$\left|\Pr\left[\mathcal{A}^{\mathsf{Sim}(x_I)}(1^\lambda) = 1\right] - \Pr\left[\mathcal{A}^{P(x,f)}(1^\lambda) = 1\right]\right|$$

is negligible in the security parameter λ, where P is an oracle that upon input (x, f) returns $\left(\pi, \{\pi_j\}_{j \in I}\right)$ with $\left(\pi, \{\pi_j\}_{j=0,\ldots,n}\right) \leftarrow \mathsf{PVP}.P(f, x)$.

We refer to [27] for the precise proving and verifying protocols and the security thereof. In combination they state a complete, sound and zero-knowledge non-interactive PVP. A prover can hence show knowledge of a sharing polynomial f to a secret $s_0 = f(0)$ with shares $s_i = f(i)$. In Section 6.1.3, we adjust [27]'s proving protocol to our setting of threshold schemes, so that knowledge of a subset of interpolation points is proven instead of all interpolation points.

Zero-Knowledge Proofs for the GAIP. We give a non-interactive zero-knowledge proof protocol for an element $s \in \mathbb{Z}_p$ with respect to the group action inverse problem. That is, a prover shows the knowledge of s so that $E_i' = [s]\,E_i$, for $E_i, E_i' \in \mathcal{E}$ and $i = 1, \ldots, m$, simultaneously, without revealing s.

The prover samples $b_j \in \mathbb{Z}_p$ and computes $\hat{E}_{i,j} \leftarrow [b_j]\,E_i$ for $i = 1, \ldots, m$ and $j = 1, \ldots, \lambda$. He then derives challenge bits

$$(c_1, \ldots, c_\lambda) \leftarrow \mathcal{H}\big(E_1, E_1', \ldots, E_m, E_m', \hat{E}_{1,1} \ldots, \hat{E}_{m,\lambda}\big)$$

via a hash function $\mathcal{H} : \mathcal{E}^{(2+\lambda)m} \to \{0,1\}^\lambda$ and prepares the answers $r_j \leftarrow b_j - c_j s$, $j = 1, \ldots, \lambda$. The proof $\pi = (c_1, \ldots, c_\lambda, r_1, \ldots, r_\lambda)$ is then published.

The verification protocol is straight forward: given a statement $(E_i, E_i')_{i=1,\ldots,m}$ and a proof $\pi = (c_1, \ldots, c_\lambda, r_1, \ldots, r_\lambda)$, the verifier computes $\tilde{E}_{i,j} \leftarrow [r_j]\,E_i$ if $c_j = 0$ and $\tilde{E}_{i,j} \leftarrow [r_j]\,E_i'$ otherwise, for $i =$

$1,\dots,m$ and $j = 1,\dots,\lambda$. He then generates verification bits $(\tilde{c}_1,\dots\tilde{c}_\lambda) \leftarrow \mathcal{H}\big(E_1,E_1',\dots,E_m,E_m',\tilde{E}_{1,1}\dots,\tilde{E}_{m,\lambda}\big)$ and accepts the proof if $(c_1,\dots,c_\lambda) = (\tilde{c}_1,\dots,\tilde{c}_\lambda)$.

We sketch the proving and verifying protocols in Algorithm 25 and Algorithm 26, respectively. Again, we refer to [28] for the proof of completeness, soundness and zero-knowledge with respect to the security parameter λ.

The Adversary. We consider a static and active adversary. At the beginning of a protocol execution, the adversary corrupts a set of shareholders. The adversary is able to see their inputs and control their outputs. The set of corrupted shareholders cannot be changed throughout the execution of the protocol.

The adversary's aim is two-fold. On the one hand it wants to obtain information on the uncorrupted parties' inputs, on the other hand it wants to falsify the output of the execution of our protocol without being detected.

Communication Channels. Both our schemes assume the existence of a trusted dealer in the secret sharing instance. The shareholders' communication occurs in the execution of the decapsulation protocol of our key exchange mechanism and the signing protocol of our signature scheme.

The communication from the dealer to a shareholder must not be eavesdropped upon or tampered with, we hence assume secure private channels between the dealer and each shareholder. However, the communication between shareholders need not be kept private, thus we assume a simple broadcast channel between the shareholders. The means of how to establish secure private channels and immutable broadcast channels are out of scope of this work.

6.1.3 Key Exchange Mechanism

A key exchange mechanism is a cryptographic public key scheme that provides three protocols: KeyGen, Encaps and Decaps. These enable a party to establish an ephemeral key between the holder of the secret key. We present our actively secure key exchange mechanism with private key that is secret shared among a set of shareholders. An authorised subset can execute the Decaps protocol with reconstructing the secret key.

Public Parameters. We fix the following publically known parameters.

- A secret sharing instance $\mathcal{S}$ with shareholders $\mathcal{S} = \{P_1,\dots,P_n\}$, secret space $\mathbb{Z}_p$ and access structure Γ.
- A hard homogeneous space $(\mathcal{E},\mathcal{G})$ with fixed starting point $E_0 \in \mathcal{E}$.
- A fixed element $g \in \mathcal{G}$ with $\mathsf{ord} g = p$ for the mapping $[\cdot]\cdot : \mathbb{Z}_p \times \mathcal{E} \to \mathcal{E}; s \mapsto g^s E$.

We give our key exchange mechanism in the context of Shamir's secret sharing scheme and elaborate possible extensions to other, more general secret sharing schemes in Section 6.1.5.

Key Generation. A public and secret key pair is established by a trusted dealer (even an untrusted dealer is feasible by employing verifiable secret sharing schemes) executing Algorithm 20. For that he samples a secret key s and publishes the public key $\mathsf{pk} \leftarrow [s]E_0$. The secret key s is then shared among the $\{P_1, \ldots, P_n\}$ via $\mathcal{S}.\mathsf{Share}(s)$. The dealer shares each share s_i, $i = 1, \ldots, n$, once more. Each shareholder P_i, $i = 1, \ldots, n$, eventually receives s_i, $\{s_{ji}, s_{ij}\}_{j=1,\ldots,n}$, that is his share s_i of s, the sharing of s_i and a share of other s_j, $j \neq i$.

Algorithm 20: Key generation

```
  Input: $\mathcal{S}$
1 $s \leftarrow\$ \mathbb{Z}_p$
2 $\mathsf{pk} \leftarrow [s] E_0$
3 $\{s_1, \ldots, s_n\} \leftarrow \mathcal{S}.\mathsf{Share}(s)$
4 for $i = 1, \ldots, n$ do
5     $\{s_{i1}, \ldots, s_{in}\} \leftarrow \mathcal{S}.\mathsf{Share}(s_i)$
6 publish $\mathsf{pk}$
7 for $i = 1, \ldots, n$ do
8     send $\left\{s_i, \{s_{ij}\}_{j=1,\ldots,n}, \{s_{ki}\}_{k=1,\ldots,n}\right\}$ to $P_i$
```

This key generation protocol can be regarded as a "two-level sharing", where each share of the secret key is itself shared again among the shareholders.

Encapsulation. With a public key $\mathsf{pk} \in \mathcal{E}$ as input, the encapsulation protocol returns an ephemeral key $\mathcal{K} \in \mathcal{E}$ and a ciphertext $c \in \mathcal{E}$. Our encapsulation protocol is identical to the protocol of [73], thus we just give a short sketch and refer to De Feo's and Meyer's work for the respective proofs of security.

Decapsulation. A decapsulation protocol takes a ciphertext c and outputs a key $\mathcal{K}$. De Feo and Meyer [73] applied the threshold group action (Algorithm 24) so that an authorised set $S' \in \Gamma$ decapsulates a ciphertext c and produces an ephemeral key $[s]c = [s](b * E_0) = b * ([s]E_0)$. For that, the shareholders agree on an arbitrary order of turns. With $E^0 := c$, for $k = 1, \ldots, \#S'$, the kth shareholder P_i outputs $E^k = [L_{i,S'}s_i]E^{k-1}$. The last shareholder outputs the decapsulated ciphertext $E^{\#S'} = [s]c$. Their approach is simulatable. It does not leak any information on the

Algorithm 21: Encapsulation

```
  Input: $\mathsf{pk}$
1 $b \leftarrow\$ \mathcal{G}$
2 $\mathcal{K} \leftarrow b * \mathsf{pk}$
3 $c \leftarrow b * E_0$
4 return $(\mathcal{K}, c)$
```

shares s_i, yet it is only passively secure. Thus, a malicious shareholder can provide malformed input to the protocol and falsify the output without being detected. We extend their approach to enable detecting misbehaving shareholders in a decapsulation. For that we maintain the threshold group action and apply the PVP and zero-knowledge proof layed out in Section 6.1.2.

Amending the PVP. In the PVP protocol sketched in Section 6.1.2, a prover produces a proof of knowledge for a witness polynomial f of the statement $((E_0, E_1), s_1, \dots, s_n)$, where $E_0 \leftarrow\$ \mathcal{E}$, $E_1 = [s_0] E_0$ and $s_i = f(i)$ for $i = 0, \dots, n$. He thereby proves knowledge of the sharing polynomial f of $s_0 = f(0)$.

This approach does not agree with the threshold group action, for which a shareholder P_i's output in the round-robin approach is $E^k \leftarrow [L_{i,S'} s_i] E^{k-1}$ rather than $E^k \leftarrow [s_i] E^{k-1}$, where E^{k-1} denotes the previous shareholder's output. Futhermore, authorised sets need not contain all shareholders. Example 11 illustrates a further conflict with of the PVP with the threshold group action.

Example 11. *Let* sk *be a secret key generated and shared by* KeyGen*. That is each shareholder* P_i *holds*

$$\left\{s_i, \{s_{ij}\}_{P_j \in S}, \{s_{ji}\}_{P_j \in S}\right\}.$$

Also let $S' \in \Gamma$ *be a minimally authorised set executing Algorithm 24, i.e., for any* $P_i \in S'$*,* $S' \setminus \{P_i\}$ *is unauthorised. Thus, for any arbitrary but fixed* $s'_i \in \mathbb{Z}_p$*, there exists a polynomial* $f'_i \in \mathbb{Z}_p[X]_{k-1}$ *so that* $f'_i(j) = L_{i,S'} s_{ij}$ *and* $R' = [f'_i(0)] R$ *for any* $R, R' \in \mathcal{E}$*. Therefore,* P_i *can publish* $\left(\pi, \{\pi_j\}_{P_j \in S'}\right)$ *with*

$$\left(\pi, \{\pi_j\}_{P_j \in S}\right) \leftarrow \mathsf{PVP}.P\Big(\left((R, R'), (L_{i,S'} s_{ij})_{P_j \in S}\right), f'_i\Big)$$

which to $S' \setminus \{P_i\}$ *is indistinguishable from*

$$\mathsf{PVP}.P\Big(\left((E_0, E_1), (L_{i,S'} s_{ij})_{P_j \in S}\right), L_{i,S'} f_i\Big)$$

with $E_0 \leftarrow\$ \mathcal{E}$ and $E_1 = [L_{i,S'} s_i] E_0$. Thus, for a minimally authorised set S', the soundness of the PVP does not hold with respect to $P_i \in S'$ and f_i.

We resolve the conflicts by amending [27]'s PVP protocol, so that, for a superauthorised set S^*, a shareholder $P_i \in S^*$ proves knowledge of a witness polynomial $L_{i,S^*} f_i$ for a statement

$$\Big((R, R'), (s_{ij})_{P_j \in S^*} \Big),$$

where $R \leftarrow\$ \mathcal{E}$, $R' = [L_{i,S^*} s_i] R$, $s_{ij} = f_i(j)$ for $P_j \in S^*$ and $s_i = f_i(0)$. The inputs of our amended proving protocol are the proving shareholder's index i, the witness polynomial f, the superauthorised set $S^* \in \Gamma^+$ and the statement $\Big((R, R'), (s_{ij})_{P_j \in S^*} \Big)$. The protocol can be found in Algorithm 27, in which $\mathcal{C}$ denotes a commitment scheme. The verifying protocol in turn has the prover's and the verifier's indices i and j, respectively, a set $S^* \in \Gamma^+$, a statement piece x_j and a proof piece (π, π_j) as input, where $x_j = (R, R') \in \mathcal{E}^2$ if $j = 0$ and $x_j \in \mathbb{Z}_p$ otherwise. The verifying protocol is given in Algorithm 28.

The definitions of soundness and zero-knowledge for a threshold PVP scheme carry over from the non-threshold setting in Section 6.1.2 intuitively, yet we restate the completeness definition for the threshold setting.

Definition 12 (Completeness in the threshold setting)**.** *We call a threshold PVP scheme* complete *if, for any $S' \in \Gamma$, any $(x, f) \in \mathcal{R}$, any $P_i \in S'$ and $\Big(\pi, \{\pi_j\}_{P_j \in S'}\Big) \leftarrow \mathsf{PVP}.P(i, f, S', x_{S'})$, we have*

$$\Pr[\mathsf{PVP}.V(i, j, S', x_j, (\pi, \pi_j)) = \mathsf{true}] = 1 \textit{ for all } P_j \in S'.$$

The proofs for soundness, correctness and zero-knowledge for the approach of [27] are easily transferred to our amended protocols, thus we do not restate them here.

We arrive at our decapsulation protocol, executed by a superauthorised set S^*: The partaking shareholders fix a turn order. A shareholder P_i's turn consists of the following steps.

1. If the previous shareholder's output E^{k-1} is not in $\mathcal{E}$, P_i outputs $\perp$ and aborts. The first shareholder's input E^0 is the protocol's input ciphertext c.
2. Otherwise P_i samples $R_k \leftarrow\$ \mathcal{E}$ and computes $R'_k \leftarrow [L_{i,S^*} s_i] R_k$.
3. P_i computes and publishes
$$\Big(\pi^k, \{\pi_j^k\}_{P_j \in S^*}\Big) \leftarrow \mathsf{PVP}.P\Big(i, f_i, S^*, \Big((R_k, R'_k), (s_{ij})_{P_j \in S^*}\Big)\Big).$$
4. P_i computes $E^k \leftarrow [L_{i,S^*} s_i] E^{k-1}$ and the zero-knowledge proof $zk \leftarrow \mathsf{ZK}.P\big((R_k, R'_k), \big(E^{k-1}, E^k\big), L_{i,S^*} s_i\big)$. He publishes both.

5. Each shareholder $P_j \in S^* \setminus \{P_i\}$ verifies
$$\mathsf{PVP}.V\big(i, j, S^*, s_{ij}, (\pi^k, \pi_j^k)\big) \wedge \mathsf{PVP}.V\big(i, 0, S^*, (R_k, R_k'), (\pi^k, \pi_0^k)\big) \tag{6.2}$$
and
$$\mathsf{ZK}.V\big((R_k, R_k'), (E^{k-1}, E^k), zk\big). \tag{6.3}$$
If (6.2) fails, P_j issues a complaint against P_i. If P_i is convicted of cheating by more than ${\#S^*}/2$ shareholders, decapsulation is restarted with an $S^{*\prime} \in \Gamma^+$, so that $P_i \notin S^{*\prime}$. If (6.3) fails, the decapsulation is restarted outright with $S^{*\prime} \in \Gamma^+$, so that $P_i \notin S^{*\prime}$.

6. Otherwise, P_i outputs E^k and finalises its turn.

7. The protocol terminates with the last shareholder's $E^{\#S^*}$ as output.

The combination of the PVP and the zero-knowledge proof in steps 3 and 4 ensure that P_i has knowledge of the sharing polynomial $L_{i,S^*} f_i$ and also inputs $L_{i,S^*} f_i(0)$ to compute E^k. We give the precise protocol in Algorithm 22.

Definition 13. *A key exchange mechanism with secret shared private key is* correct*, if for any authorised set S', any public key* pk *and any* $(\mathcal{K}, c) \leftarrow \mathsf{Encaps}(\mathsf{pk})$*, we have* $\mathcal{K} = \mathcal{K}' \leftarrow \mathsf{Decaps}(c, S')$.

The correctness of our key exchange mechanism presented in Algorithm 20, Algorithm 21 and Algorithm 22 follows from the correctness of the threshold group action (Algorithm 24). Let sk be a secret key and $\mathsf{pk} = [\mathsf{sk}]\, E_0$ be the respective public key, that have been generated by KeyGen, thus each shareholder P_i holds a share s_i of sk, $i = 1, \ldots, n$. For an authorised set S' we therefore have
$$\mathsf{sk} = \sum_{P_i \in S'} L_{i,S'} s_i.$$
Furthermore, let $(\mathcal{K}, c) \leftarrow \mathsf{Encaps}(\mathsf{pk})$. To show correctness, $\mathcal{K}' = \mathcal{K}$ has to hold, where $\mathcal{K}' \leftarrow \mathsf{Decaps}(c, S')$. Now, after executing $\mathsf{Decaps}(c, S')$, we have $\mathcal{K}' = E^{\#S'}$ emerging as the result of the threshold group action applied to c. This gives us
$$\mathcal{K}' = \left[\sum_{P_i \in S'} L_{i,S'} s_i\right] c = [\mathsf{sk}]\,(b * E_0) = b * \mathsf{pk} = \mathcal{K}.$$
The decapsulation is executed by superauthorised sets $S^* \in \Gamma^+ \subset \Gamma$. This shows that our key exchange mechanism is correct.

Algorithm 22: Decapsulation

```
Input: $c, S^*$
$E^0 \leftarrow c$
$k \leftarrow 0$
for $P_i \in S^*$ do
    if $E^k \notin \mathcal{E}$ then
        $P_i$ outputs $\perp$ and aborts.
    $k \leftarrow k + 1$
    $R_k \leftarrow\$ \mathcal{E}$
    $R'_k \leftarrow [L_{i,S^*} s_i] R_k$
    $\left(\pi^k, \{\pi_j^k\}_{P_j \in S^*}\right) \leftarrow \mathsf{PVP}.P\left(i, f_i, S^*, ((R_k, R'_k), (s_{ij})_{P_j \in S^*})\right)$
    $P_i$ publishes $(R_k, R'_k)$ and $\left(\pi^k, \{\pi_j^k\}_{P_j \in S^*}\right)$
    $E^k \leftarrow [L_{i,S^*} s_i] E^{k-1}$
    $zk^k \leftarrow \mathsf{ZK}.P\big((R_k, R'_k), \left(E^{k-1}, E^k\right), L_{i,S^*} s_i\big)$
    $P_i$ publishes $\left(E^k, zk^k\right)$
    for $P_j \in S^* \setminus \{P_i\}$ do
        if $\mathsf{PVP}.V\big(i, j, S^*, s_{ij}, \left(\pi^k, \pi_j^k\right)\big) = \mathsf{false} \vee$
         $\mathsf{PVP}.V\big(i, 0, S^*, (R_k, R'_k), \left(\pi^k, \pi_0^k\right)\big) = \mathsf{false}$ then
            $P_j$ publishes $s_{ij}$
            if $P_i$ is convicted then
                return $\mathsf{Decapsulation}\big(c, S^{*\prime}\big)$ with $S^{*\prime} \in \Gamma \wedge P_i \notin S^{*\prime}$
        if $\mathsf{ZK}.V\big((R_k, R'_k), \left(E^{k-1}, E^k\right), zk\big) = \mathsf{false}$ then
            return $\mathsf{Decapsulation}\big(c, S^{*\prime}\big)$ with $S^{*\prime} \in \Gamma \wedge P_i \notin S^{*\prime}$
return $\mathcal{K} \leftarrow E^k$
```

Security. There are two aspects of security to consider:

- Active security: A malicious shareholder cannot generate his contribution to the decapsulation protocol dishonestly without being detected. We prove this by showing that an adversary that can provide malformed inputs without detection can break the PVP or the zero-knowledge proof of knowledge.
- Simulatability: An adversary that corrupts an unauthorised set of shareholders cannot learn any information about the uncorrupted shareholders' inputs from an execution of the decapsulation protocol. We show this by proving the simulatability of Decaps.

Active security.

Theorem 14. *Let $S^* \in \Gamma^+$ and let* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}$ *be a public/secret key pair, where* sk *has been shared. Also let* $(\mathcal{K}, c) \leftarrow \mathsf{Encaps}(\mathsf{pk})$. *Denote the transcript of* $\mathsf{Decaps}(c, S^*)$ *by*

$$\left(E^k, (R_k, R'_k), \left(\pi^k, \{\pi_j^k\}_{P_j \in S^*}\right), zk_k\right)_{k=1,\dots,\#S^*}.$$

Let $P_i \in S^*$ be an arbitrary but fixed shareholder. If $\mathsf{Decaps}(c, S^*)$ *terminated successfully and $P_{i'}$'s output was generated dishonestly, then there exists an algorithm that breaks the soundness property of* PVP *or* ZK.

Proof. Let $P_{i'}$ be the malicious shareholder and let k' be the index of $P_{i'}$'s output in the transcript. Since $\mathsf{Decaps}(c, S^*)$ terminated successfully, we have

$$\mathsf{PVP}.V\left(i', j, S^*, L_{i',S^*} s_{i'j}, \left(\pi^{k'}, \pi_j^{k'}\right)\right) = \mathsf{true} \tag{6.4}$$

$$\mathsf{PVP}.V\left(i', 0, S^*, (R_{k'}, R'_{k'}), \left(\pi^{k'}, \pi_0^{k'}\right)\right) = \mathsf{true} \tag{6.5}$$

$$\mathsf{ZK}.V\left(\left(E^{k'-1}, E^{k'}\right), (R_{k'}, R'_{k'}), zk^{k'}\right) = \mathsf{true} \tag{6.6}$$

for all $P_j \in S^* \setminus \{P_{i'}\}$. $E^{k'}$ was generated dishonestly, thus we have

$$E^{k'} = [\alpha] E^{k'-1}, \text{ for some } \alpha \neq L_{i',S^*} s_{i'}.$$

We distinguish two cases: $R'_{k'} \neq [\alpha] R_{k'}$ and $R'_{k'} = [\alpha] R_{k'}$.

In the first case, $P_{i'}$ published a zero-knowledge proof $zk^{k'}$ so that (6.6) holds, where $E^{k'} = [\alpha] E^{k'-1}$ yet $R'_{k'} \neq [\alpha] R_{k'}$. $P_{i'}$ thus broke the soundness property of the zero-knowledge proof.

In the second case, $P_{i'}$ published $\left(\pi^{k'}, \left\{\pi_j^{k'}\right\}_{P_j \in S^*}\right)$ so that (6.4) and (6.5) hold for all $P_j \in S^* \setminus \{P_{i'}\}$ and for $j = 0$. Thus, $P_{i'}$ proved knowledge of a witness polynomial f' with

$$f'(j) = L_{i',S^*} s_{ij} \tag{6.7}$$

for all $P_j \in S^* \setminus \{P_{i'}\}$ and $R'_{k'} = [f'(0)] R_{k'}$, that is $f'(0) = \alpha$. Since f' has degree at most $k-1$, it is well-defined from (6.7). Thus, we have $f' \equiv L_{i',S^*} f_{i'}$, where $f_{i'}$ is the polynomial with which s_i was shared, i.e., $f_{i'}(0) = s_i$. This gives us $\alpha = f'(0) = L_{i',S^*} f_{i'}(0) = L_{i',S^*} s_{i'j}$. We arrive at a contradiction, assuming the soundness of the PVP. □

Simulatability. We show, that an adversary who corrupts an unauthorised subset of shareholder does not learn any additional information from an execution of the decapsulation protocol.

Definition 15 (Simulatability). *We call a key exchange mechanism* simulatable*, if for any HHS $(\mathcal{E},\mathcal{G})$ with security parameter λ and any compatible secret sharing instance $\mathcal{S}$, there exists a polynomial-time algorithm* Sim *so that, for any polynomial-time adversary $\mathcal{A}$ the advantage*

$$\mathsf{Adv}^{\text{dist-transcript}}_{\mathcal{A},\mathsf{Sim}}((\mathcal{E},\mathcal{G}),\mathcal{S}) := \left|\Pr\left[\mathsf{Exp}^{\textit{dist-transcript}}_{\mathcal{A},\mathsf{Sim}}(\mathcal{S})\right]\right|$$

in the security game $Exp^{\textit{dist-transcript}}_{\mathcal{A},\mathsf{Sim}}(\mathcal{S})$ (Algorithm 29) is negligible in λ.

Theorem 16. *If the* PVP *protocol and the GAIP* ZK *protocol employed are zero-knowledge, then the decapsulation protocol (Algorithm 22) is simulatable.*

Proof. We give a finite series of simulators, the first of which simulates the behaviour of the uncorrupted parties faithfully and the last of which fulfills the secrecy requirements. This series is inspired by the simulators, that [27] gave for the secrecy proof of their key generation algorithm, yet differs in some significant aspects. The outputs of the respective simulators will be proven indistinguishable, hence resulting in the indistinguishability of the first and last one. As a slight misuse of the notation, we denote the set of corrupted shareholders by $\mathcal{A}$, where $\mathcal{A}$ is the adversary corrupting an unauthorised set of shareholders. This means P_i is corrupted iff $P_i \in \mathcal{A}$.

The input for each simulator is a ciphertext c, a derived key $\mathcal{K}$ and the adversary's knowledge after KeyGen was successfully executed, that is

$$\left\{s_i, \{s_{ij}\}_{P_i\in S^*}, \{s_{ji}\}_{P_j\in S^*\setminus\mathcal{A}}\right\}_{P_i\in\mathcal{A}}.$$

1. The adversary corrupted an unauthorised set $\mathcal{A}$, hence each share of the secret key is uniformly distributed from his view. Sim^1 samples a polynomial $f_i' \in \mathbb{Z}_p[X]_{k-1}$ with $\forall P_j \in \mathcal{A} : f_i'(i) = s_{ij}$ uniformly at random for each $P_i \in S^* \setminus \mathcal{A}$. Since $\mathcal{A}$ is unauthorised, f_i' exists.

   Sim^1 then proceeds by honestly producing the output of each $P_i \in S^* \setminus \mathcal{A}$ according to the decapsulation protocol, i.e., it samples $R_k \leftarrow\$ \mathcal{E}$, computes $R_k' \leftarrow [L_{i,S^*} f_i'(0)]\, R_k$ and outputs

   $$\mathsf{PVP}.P\Big(i, f_i', S^*, \Big((R_k, R_k'), (L_{i,S^*} s_{ij})_{P_j\in S^*}\Big)\Big),$$

   $E^k \leftarrow [L_{i,S^*} s_i']\, E^{k-1}$ and $\mathsf{ZK}.P\big((R_k, R_k'), (E^{k-1}, E^k), L_{i,S^*} f_i'(0)\big)$, where k is the index of P_i's output in the transcript, $s_{ij} := f_i'(j)$ for $P_j \in S^* \setminus \mathcal{A}$ and $s_i' := f_i'(0)$. Since, for all $P_i \in S^* \setminus \mathcal{A}$, s_i is information theoretically hidden to the adversary, the resulting transcript is identically distributed to a real transcript.

2. Let i' denote the index of the last honest party in the execution of the decapsulation protocol and k' the index of its output. Sim^2 behaves

exactly as Sim^1 with the exception, that it does not compute the PVP itself but calls the simulator $\mathsf{Sim}^{\mathsf{PVP}}$ for the PVP to generate the proof $\left(\pi^{k'}, \left\{\pi_j^{k'}\right\}\right)$ for the statement $\left((R_{k'}, R'_{k'}), (L_{i,S^*} s_{i'j})_{P_j \in S^*}\right)$. Since the PVP is zero-knowledge, Sim^2's output is indistinguishable from that of Sim^1.

3. Sim^3 behaves identical to Sim^2 apart from not generating the zero-knowledge proof for $P_{i'}$ itself, but outsourcing it to the simulator for the zero-knowledge proof. That is Sim^3 hands tuples $(R_{k'}, R'_{k'})$ and $\left(E^{k'-1}, E^{k'}\right)$ to $\mathsf{Sim}^{\mathsf{ZK}}$ and publishes its answer as the zero-knowledge proof. With ZK being zero-knowledge, the output of Sim^3 is indistinguishable from that of Sim^2.

4. The final simulator, Sim^4, enforces the correct decapsulation output, that is $E^{\#S^*} = \mathcal{K}$. Since, for $P_j \in \mathcal{A}$, s_j was provided as input and $P_{i'}$ is the last honest shareholder in the order of decapsulation execution, Sim^4 computes
$$\sum_{P_j \in S'} L_{j,S^*} s_j,$$
where S' contains the shareholders, whose turn is after $P_{i'}$'s. To achieve the correct output of the decapsulation E, Sim^4 thus sets
$$E^{k'} \leftarrow \left[- \sum_{P_j \in S'} L_{j,S^*} s_j \right] E$$
instead of $E^{k'} \leftarrow [L_{i',S^*} s'_{i'}] E^{k'-1}$. Assuming the soundness of the PVP as well as of the zero-knowledge proof, this guarantees the result to be $E^{\#S^*} = E$, since
$$E^{\#S^*} = \left[\sum_{P_j \in S'} L_{j,S^*} s_j \right] E^{k'} = E$$
holds. It remains to show, that the output of Sim^4 cannot be distinguished from that of Sim^3. The following reasoning is similar to that of [27], yet for completeness we give a reduction $\mathcal{B}'$, that uses a distinguisher $\mathcal{A}'$, that distinguishes Sim^3 from Sim^4, to break the decisional parallelisation problem. We highlight the necessary modifications.

Let (E_a, E_b, E_c) be an instance of the decisional parallelisation problem with base element c. $\mathcal{B}'$ computes
$$E^{k'} \leftarrow \left[\sum_{P_j \in S^* \setminus (S' \cup \{P_{i'}\})} L_{j,S^*} s_j \right] E_a.$$

With $s_{i'}$ looking uniformly distributed from $\mathcal{A}$'s view, this choice of $E^{k'}$ is indistinguishable from $E^{k'} = [L_{i',S^*} s'_{i'}] E^{k'-1}$. $\mathcal{B}'$ furthermore does not sample $R_{k'} \leftarrow\$ \mathcal{E}$ but puts $R_{k'} \leftarrow E_b$ and $R'_{k'} \leftarrow E_c$. The resulting transcript is handed to $\mathcal{A}'$ and $\mathcal{B}'$ outputs whatever $\mathcal{A}'$ outputs.

Comparing the distributions, we see that

$$E^{k'} = [a] E^{k'-1} = [a] \left(\left[\sum_{P_j \in S^* \setminus (S' \cup \{P_{i'}\})} L_{j,S^*} s_j \right] c \right)$$

if and only if $E_a = [a]c$, where $s_j := s'_j$ for $P_j \notin \mathcal{A}$. Furthermore, $R'_{k'} = [a]R_{k'}$ is equivalent to $E_c = [a]E_b$. In the case of $E_a = [a]c$ and $E_c = [a]E_b$, the transcript handed to $\mathcal{A}'$ is identically distributed to Sim^3's output. If, on the other hand, (E_a, E_b, E_c) is a random triple, then the transcript follows the same distribution as Sim^4's output. $\mathcal{B}'$ thus breaks the DPP with the same advantage as $\mathcal{A}'$ distinguishes Sim^3 from Sim^4.

Sim^4 outputs a transcript of the decapsulation protocol with input c and output $\mathcal{K}$ that cannot be distinguished from the output of Sim^1, which is indistinguishable from a real execution protocol. □

Efficiency. Each shareholder engaged in an execution of the decapsulation protocol has one round of messages to send. The messages of the k-th shareholder consist of the tuple (R_k, R'_k), a PVP proof $\left(\pi^k, \{\pi^k_j\}_{P_j \in S^*}\right)$, the output E^k and the zero-knowledge proof zk. Thus, the total size of a shareholder's messages is

$$\begin{aligned} &2x + 2c + \lambda k \log p + 2\lambda(\#S^*) + x + \lambda k \log p + \lambda \\ =&3x + 2c + \lambda\left(1 + 2(\#S^*) + 2k \log p\right) \end{aligned}$$

where x is the size of the bit representation of an element of $\mathcal{E}$ and c is the size of a commitment produced in $\mathsf{PVP}.P$. Assuming x, c and the secret sharing parameters k and p to be constant, the message size is thus linear in the security parameter λ with moderate cofactor.

6.1.4 Actively Secure Secret Shared Signature Protocols

We convert the key exchange mechanism in Algorithm 20, Algorithm 21 and Algorithm 22 into an actively secure signature scheme with secret shared signing key. We concede that applying active security measures to a signature scheme to ensure the correctness of the resulting signature is counterintuitive, since the correctness of a signature can easily be checked through the verifying protocol. Yet verification returning false only shows that the signature is incorrect, a misbehaving shareholder cannot be identified this way. An actively secure signature scheme achieves just that. An identified cheating shareholder can hence be excluded from future runs of the signing protocol.

A signature scheme consists of three protocols: key generation, signing and verifying. We transfer the unmodified key generation protocol from the key exchange mechnism in Section 6.1.3 to our signature scheme. The signing protocol is derived from the decapsulation protocol (Algorithm 22) by applying the Fiat-Shamir-transformation, the verifying protocol follows straightforward. The protocols are given in Algorithm 23 and Algorithm 30.

Similar to [28], the results from [78] on Fiat-Shamir in the QROM can be applied to our setting as follows. First, in the case without hashing, since the sigma protocol has special soundness [28] and in our case perfect unique reponses, [78] shows that the protocol is a quantum proof of knowledge. Further, in the case with hashing, the collapsingness property implies that the protocol has unique responses in a quantum scenario.

Instantiations. As a practical instantiation, we propose the available parameter set for CSIDH-512 HHS from [28]. Currently no other instantiation of the presented schemes seems feasible in a practical sense. Furthermore, according to recent works [158, 35] CSIDH-512 may not reach the initially estimated security level.

6.1.5 Generalising the Secret Sharing Schemes

We constructed the protocols above in the context of Shamir's secret sharing protocol [168]. The key exchange mechanism in Section 6.1.3 as well as the signature scheme in Section 6.1.4 can be extended to more general secret sharing schemes. In the following, we characterise the requirements that a secret sharing scheme has to meet in order to successfully implement the key exchange mechanism and the signature scheme.

Compatibility Requirements.

Definition 29 (Independent Reconstruction). *We say a secret sharing instance $\mathcal{S} = (S, \Gamma, G)$ is* independently reconstructible*, if, for any shared secret $s \in G$, any $S' \in \Gamma$ and any shareholder $P_i \in S'$, P_i's input to reconstructing s is independent of the share of each other engaged shareholder $P_j \in S'$.*

A secret sharing scheme compatible with our key exchange mechanism and signature scheme has to be independently reconstructible, since each shareholder's input into the threshold group action is hidden from every other party by virtue of the GAIP.

Algorithm 23: Secret Shared Signing Algorithm

```
   Input: $m, S^*$
1  $\left(E_1^0, \dots, E_\lambda^0\right) \leftarrow (E_0, \dots, E_0)$
2  $k \leftarrow 0$
3  for $P_i \in S^*$ do
4      $k \leftarrow k + 1$
5      for $l \in 1, \dots, \lambda$ do
6          $P_i$ samples $b_{il} \leftarrow\$ \mathbb{Z}_q[X]_{\leq k-1}$
7          $P_i$ publishes $R_{il}^k \leftarrow\$ \mathcal{E}$
8          $P_i$ publishes ${R'_{il}}^k \leftarrow [b_{il}(0)]\, R_{il}^k$
9          $P_i$ publishes $\left(\pi, \{\pi_j\}_{P_j \in S^*}\right) \leftarrow$
             $\mathsf{PVP}.P\left(i, b_{il}, S^*, \left(\left(R_{il}^k, {R'_{il}}^k\right), (b_{il}(l))_{P_j \in S^*}\right)\right)$
10         $P_i$ outputs $E_l^k \leftarrow [b_{il}(0)]\, E_l^{k-1}$
11         $P_i$ publishes $zk \leftarrow \mathsf{ZK}.P\left(\left(R_{il}^k, {R'_{il}}^k\right), \left(E_l^{k-1}, E_l^k\right), b_{il}(0)\right)$
12         if $\mathsf{ZK}.V\left(\left(R_{il}^k, {R'_{il}}^k\right), \left(E_l^{k-1}, E_l^k\right), zk\right) = \mathsf{false}$ then
13             restart without $P_i$

14 $(c_1, \dots, c_\lambda) \leftarrow \mathcal{H}\left(E_1^{\#S^*}, \dots, E_\lambda^{\#S^*}, m\right)$
15 for $P_i \in S^*$ do
16     for $l \in 1, \dots, \lambda$ do
17         $P_i$ outputs $z_{il} = b_{il} - c_l \cdot L_{i,S^*} \cdot s_i$
18         for $P_j \in S^*$ do
19             $P_j$ computes $b'_{il}(j) \leftarrow z_{il}(j) + c_l L_{i,S^*} s_{ij}$
20             and verifies
21             $\mathsf{PVP}.V(i, j, S^*, b'_{il}(j), \pi, \pi_j) \wedge$
                 $\mathsf{PVP}.V\left(i, 0, S^*, \left(R_{il}^k, {R'_{il}}^k\right), \pi, \pi_0\right)$
22             if $P_i$ *is convicted of cheating* then
23                 restart without $P_i$

24 for $l \in 1, \dots, \lambda$ do
25     $z_j \leftarrow \sum_{P_i \in S^*} z_{ij}$
26 return $((c_1, \dots, c_\lambda), (z_1, \dots, z_\lambda))$
```

Definition 30 (Self-contained reconstruction)**.** *An instance $\mathcal{S} = (S, \Gamma, G)$ of a secret sharing scheme is called* self-contained*, if, for any authorised set S', the input of any shareholder $P_i \in S'$ in an execution of* Rec *is an element of G.*

It is necessary, that $G = \mathbb{Z}_p$ for some prime p holds to enable the mapping $\cdot \mapsto [\cdot]$. This requirement may be loosened by replacing $\cdot \mapsto [\cdot]$ appropriately. To enable two-level sharing, it has to hold that for a share $s_i \in \mathcal{S}.\mathsf{Share}(s)$ of a secret s, $s_i \in G$ holds. The secret sharing scheme also has to allow for a PVP scheme, that is compatible with a zero-knowledge proof for the GAIP.

Examples of Secret Sharing Schemes.

- It is evident, that Shamir's approach fulfills all aforementioned requirements. In fact, the two-level sharing and the PVP have been tailored to Shamir's polynomial based secret sharing approach.

- Tassa [176] extended Shamir's approach of threshold secret sharing to a hierarchical access structure. To share a secret $s \in \mathbb{Z}_p$ with prime p, a polynomial f with constant term s is sampled. Shareholders of the top level of the hierarchy are assigned interpolation points of f as in Shamir's scheme. The k-th level of the hierarchy receives interpolation points of the $k-1$st derivative of f. The shares of in Tassa's scheme are elements of $\mathbb{Z}_p$ themselves. The key generation (Algorithm 20) must be adapted so that a shareholder receives a description of the polynomial utilised in sharing his share, instead of receiving the shares with which his share of the secret key was shared. Hence all derivatives utilised can easily be computed. Reconstructing a shared secret is achieved via Birkhoff interpolation, the execution of which is independent and self-contained. The zero-knowledge proof (Algorithm 25 and Algorithm 26) as well as the piecewise verifiable proof (Algorithm 27 and Algorithm 28) thus directly transfer to Tassa's approach utilising the appropriate derivatives in the verifying protocols. The decapsulation and the signing protocols hence can be executed with adjustments only to the verifying steps.

- In 2006, Damgard and Thorbek proposed a linear integer secret sharing scheme [69] with secret space $\mathbb{Z}$. Given an access structure Γ, a matrix M is generated in which each shareholder is assigned a column so that iff $S' \in \Gamma$, the submatrix $M_{S'}$ has full rank. A secret s is shared by multiplying a random vector v with first entry s with M and sending the resulting vector entries to the respective shareholders. Reconstruction follows intuitively. Their scheme hence further generalises Tassa's with respect to secret space and feasible access structures. With the secret space $\mathbb{Z}$ their approach is not compatible with the mapping $\cdot \mapsto [\cdot]$ and our PVP scheme. Thus, neither our key exchange mechanism nor our signature scheme can in its current form be instantiated with Damgard's and Thorbek's scheme.

6.1.6 Conclusion

In this work, we presented an actively secure key exchange mechanism based on Shamir's secret sharing scheme and derived a signature scheme from it. The active security measures consist of a piecewise verifiable proof and a zero-knowledge proof for the GAIP, that in combination prove the knowledge of the correct share of the secret key and ensure its use in the protocol. For that we reworked the piecewise verifiable proof and zero-knowledge proof introduced in [27] to fit the threshold setting of Shamir's secret sharing and applied it to the threshold group action of [73]. Active security and simulatability were proven under the assumption of hardness of the decisional parallelisation problem.

Furthermore, we characterised the properties necessary for a secret sharing scheme in order for our key exchange mechanism and signature scheme to be based on it. We gave examples and counter-examples of secret sharing schemes compatible with our approach to demonstrate its limits. We thereby demonstrated that cryptographic schemes with secret shared private key in the HHS setting are not limited to threshold schemes, but applicable to more general access structures.

6.1.7 Appendix

6.1.7.1 Algorithms

Algorithm 24: Threshold group action

```
   Input: $E, S'$
1  $E^0 \leftarrow E$
2  $k \leftarrow 0$
3  for $P_i \in S'$ do
4      if $E^k \notin \mathcal{E}$ then
5          $P_i$ outputs $\perp$ and aborts.
6      else
7          $k \leftarrow k + 1$
8          $P_i$ outputs $E^k \leftarrow [L_{i,S'} s_i] E^{k-1}$
9  return $E^k$
```

Algorithm 25: The ZK proving protocol for the GAIP

```
  Input: $s, (E_i, E_i')_{i=1,\dots,m}$
1 for $j = 1, \dots, \lambda$ do
2     $b_j \leftarrow\$ \mathbb{Z}_p$
3     for $i = 1, \dots, m$ do
4         $\hat{E}_{ij} \leftarrow [b_j] E_i$
5 $(c_1, \dots, c_\lambda) \leftarrow \mathcal{H}\big(E_1, E_1', \dots, E_m, E_m', \hat{E}_{1,1}, \dots, \hat{E}_{m,\lambda}\big)$
6 for $j = 1, \dots, m$ do
7     $r_j \leftarrow b_j - c_j s$
8 return $\pi \leftarrow (c_1, \dots, c_\lambda, r_1, \dots, r_\lambda)$
```

Algorithm 26: The ZK verifying protocol for the GAIP

```
  Input: $\pi, (E_i, E_i')_{i=1,\dots,m}$
1 Parse $(c_1, \dots, c_\lambda, r_1, \dots, r_\lambda) \leftarrow \pi$
2 for $i = 1, \dots, m$ and $j = 1, \dots, \lambda$ do
3     if $c_j == 0$ then
4         $\tilde{E}_{i,j} \leftarrow [r_j] E_i$
5     else
6         $\tilde{E}_{i,j} \leftarrow [r_j] E_i'$
7 $(c_1', \dots, c_\lambda') \leftarrow \mathcal{H}\big(E_1, E_1', \dots, E_m, E_m', \tilde{E}_{1,1}, \dots, \tilde{E}_{m,\lambda}\big)$
8 return $(c_1, \dots, c_\lambda) == (c_1', \dots, c_\lambda')$
```

Algorithm 27: Proving protocol of the threshold PVP

```
   Input: $i, f, S^*, ((E_0, E_1), (s_{ij})_{P_j \in S^*})$
1  for $l \in 1, \ldots, \lambda$ do
2      $b_l \leftarrow\$ \mathbb{Z}_N[x]_{\leq k-1}$
3      $\hat{E}_l \leftarrow [b_l(0)]\, E_0$
4  $y_0, y_0' \leftarrow\$ \{0,1\}^\lambda$
5  $C_0 \leftarrow \mathcal{C}\big(\hat{E}_1 \| \ldots \| \hat{E}_\lambda, y_0\big)$
6  $C_0' \leftarrow \mathcal{C}(E_0 \| E_1, y_0')$
7  for $P_j \in S^*$ do
8      $y_j, y_j' \leftarrow\$ \{0,1\}^\lambda$
9      $C_j \leftarrow \mathcal{C}(b_1(j) \| \ldots \| b_\lambda(j), y_j)$
10     $C_j' \leftarrow \mathcal{C}\big(L_{i,S^*} \cdot s_{ij}, y_j'\big)$
11 $C \leftarrow (C_j)_{P_j \in S^*}$
12 $C' \leftarrow \big(C_j'\big)_{P_j \in S^*}$
13 $c_1, \ldots, c_\lambda \leftarrow \mathcal{H}(C, C')$
14 for $l \in 1, \ldots, \lambda$ do
15     $r_l \leftarrow b_l - c_l \cdot L_{i,S^*} \cdot f$
16 $\mathbf{r} \leftarrow (r_1, \ldots, r_\lambda)$
17 $\Big(\pi, \{\pi_j\}_{P_j \in S^*}\Big) \leftarrow \Big((C, C', \mathbf{r}), \big\{\big(y_j, y_j'\big)\big\}_{P_j \in S^*}\Big)$
18 return $\Big(\pi, \{\pi_j\}_{P_j \in S^*}\Big)$
```

Algorithm 28: Verifying protocol of the threshold PVP

```
   Input: $i, j, S^*, x_j, (\pi, \pi_j)$
1  parse $(C, C', \mathbf{r}) \leftarrow \pi$
2  parse $(y_j, y'_j) \leftarrow \pi_j$
3  $c_1, \ldots, c_\lambda \leftarrow \mathcal{H}(C, C')$
4  if $j == 0$ then
5      if $C'_j \neq \mathcal{C}(x_j, y'_j)$ then
6          return false
7      for $l \in 1, \ldots, \lambda$ do
8          $\tilde{E}_l \leftarrow [r_l(0)]\, E_{c_l}$
9      return $C_0 == \mathcal{C}\big(\tilde{E}_1 \| \ldots \| \tilde{E}_\lambda, y_0\big)$
10 else
11     if $C'_j \neq \mathcal{C}\big(L_{i,S^*} x_j, y'_j\big)$ then
12         return false
13     return $C_j == \mathcal{C}(r_1(j) + c_1 \cdot L_{i,S^*} \cdot x_j \| \ldots \| r_\lambda(j) + c_\lambda \cdot L_{i,S^*} \cdot x_j, y_j)$
```

Algorithm 29: The security game $\mathsf{Exp}^{\text{dist-transcript}}_{\mathcal{A},\mathsf{Sim}}(S)$

```
Input: $\mathcal{S}$
1  $b \leftarrow\!\!\$ \{0,1\}$
2  $S^* \leftarrow\!\!\$ \Gamma^+$
3  $S' \leftarrow\!\!\$ 2^{S^*} \setminus \Gamma$
4  $\Big(\{s_i, \{s_{ij}\}, \{s_{ji}\}\}_{P_i, P_j \in S}, \mathsf{pk}\Big) \leftarrow \mathsf{KeyGen}(\mathcal{S})$
5  $(\mathcal{K}, c) \leftarrow \mathsf{Encaps}(\mathsf{pk})$
6  $t_0 \leftarrow \mathsf{Sim}\Big(\mathcal{K}, c, \{s_i, \{s_{ij}\}, \{s_{ji}\}\}_{P_i \in S', P_j \in S}\Big)$
7  $E^0 \leftarrow E_0$, $k \leftarrow 0$
8  for $P_i \in S^*$ do
9      $k \leftarrow k + 1$
10     $E^k \leftarrow [L_{i,S^*} s_i]\, E^{k-1}$
11     $R_k \leftarrow\!\!\$ \mathcal{E}$
12     $R'_k \leftarrow [L_{i,S^*} s_i]\, R_k$
13     $\left(\pi^k, \{\pi^k_j\}_{P_j \in S^*}\right) \leftarrow \mathsf{PVP}.P\left(i, f_i, S^*, ((R_k, R'_k), (L_{i,S^*} s_{ij})_{P_j \in S^*})\right)$
14     $zk^k \leftarrow \mathsf{ZK}.P\Big((R_k, R'_k), \left(E^{k-1}, E^k\right), L_{i,S^*} s_i\Big)$
15 $t_1 \leftarrow \left(E^k, \left(\pi^k, \{\pi^k_j\}_{P_j \in S^*}\right), zk^k\right)_{k=1,\ldots,\#S^*}$
16 $b' \leftarrow \mathcal{A}(t_b)$
17 return $b == b'$
```

Algorithm 30: Signature verification protocol

```
Input: $m, s, \mathsf{pk}$
1  parse $(c_1, \ldots, c_\lambda, z_1, \ldots, z_\lambda) \leftarrow s$
2  for $j = 1, \ldots, \lambda$ do
3      if $c_j == 0$ then
4          $E'_j \leftarrow [z_j]\, E_0 = \left[\sum_{P_i \in S^*} b_{ij}\right] E_0$
5      else
6          $E'_j \leftarrow [z_j]\, \mathsf{pk} = \left[\sum_{P_i \in S^*} b_{ij} - L_{i,S^*} s_i + s\right] E_0$
7  $(c'_1, \ldots, c'_\lambda) \leftarrow \mathcal{H}(E'_1, \ldots, E'_\lambda, m)$
8  return $(c_1, \ldots, c_\lambda) == (c'_1, \ldots, c'_\lambda)$
```

7 Outlook

In this chapter, we shortly discuss some recent and a few lines of possible future work, some already under investigation.

Masking SIDH

Castryck and Decru presented in [46] an efficient key recovery algorithm against SIDH along with code demonstrating its practicality. Although their work relies on the particular choice of starting curve, the work by Maino and Martindale [132] and Robert [163] has shown that any modification of SIDH regarding the starting curve does not lead to a secure scheme. Some countermeasures [143, 85] have been proposed to avoid the Castryck and Decru family of attacks by masking the degree of the isogenies or the torsion point images, respectively. Somehow, both approaches significantly degrade the performance and key size of SIDH by at least an order of magnitude. We emphasize that some isogeny-based schemes, such as CSIDH and variants [49, 12], and SQIsign [84], are not based on SIDH, and thus, not harmed by these attacks. Thus, the focus should be on the further development of these schemes.

PQC standardization process

In 2022, after selecting CRYSTALS-KYBER for public-key encryption and key-establishment, and CRYSTALS-DILITHIUM, Falcon, and SPHINCS+ for digital signatures, NIST announced that the standardization process is continuing with a fourth round. Thereby, four of the alternate key-establishment candidate algorithms will be considered to a fourth round of evaluation. Since there are no remaining digital signature candidates under consideration, NIST is calling[38] for additional digital signature proposals.

[38] https://csrc.nist.gov/projects/pqc-dig-sig

Thereby, NIST is primarily interested on diversifying the signature standards. For this, NIST is mainly interested in signature schemes that are not based on structured lattices.

Some major threads for further research related to the selected standards are:

- computer-aided formal verification,
- secure implementation and optimization on different architectures, and
- further cryptanalysis.

NIKE vs KEM

CSIDH is currently the only option for a post-quantum drop-in replacement for Diffie–Hellman key exchange. Therefore, a promising area for future research is to understand the impact of the lower bandwidth due to the smaller number of roundtrips of a CSIDH-based NIKE compared to, for example, KEMTLS [166].

Further, since several security analyses [35, 158] suggests that CSIDH may not reach the post-quantum security by the initially proposed parameters, future research should focus on how to further improve the performance of CSIDH.

Formal verification

Evidently [116], today's methods for designing, implementing, and deploying cryptographic mechanisms are prone to introduce vulnerabilities. Usually, the verification whether cryptographic constructions satisfy desired properties is done based on handwritten proofs. As complexity increases in the field of cryptography, so does the complexity of the cryptographic constructions and their proofs.

Computer-aided cryptography [14] achieves stronger assurances regarding the correctness of implementations. Frameworks like Jasmin [5] provides both high-level structured control flow and low-level idioms for writing high-assurance and high-speed cryptography.

Considering the devasting effects of, e.g., micro-architectural attacks [117, 128], recently selected cryptographic algorithms for standardization should provide formally verified implementations [6, 169]. Therefore, future standardization process should require computer-verified proofs at some point during the evaluation process.

Bibliography

[1] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 322–343. Springer, 2018. https://doi.org/10.1007/978-3-030-10970-7_15.

[2] Gora Adj, Jesús-Javier Chi-Domínguez, Víctor Mateu, and Francisco Rodríguez-Henríquez. Faulty isogenies: a new kind of leakage. *IACR Cryptol. ePrint Arch.*, page 153, 2022. https://eprint.iacr.org/2022/153.

[3] Gora Adj, Jesús-Javier Chi-Domínguez, and Francisco Rodríguez-Henríquez. On new Vélu's formulae and their applications to CSIDH and B-SIDH constant-time implementations, 2020. https://eprint.iacr.org/2020/1109.

[4] Toru Akishita and Tsuyoshi Takagi. Zero-value point attacks on elliptic curve cryptosystem. In Colin Boyd and Wenbo Mao, editors, *Information Security, 6th International Conference, ISC 2003, Bristol, UK, October 1-3, 2003, Proceedings*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2003. https://doi.org/10.1007/10958513_17.

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*,

pages 1807–1823. ACM, 2017. https://doi.org/10.1145/3133956.3134078.

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying kyber episode iv: Implementation correctness. Cryptology ePrint Archive, Paper 2023/215, 2023. https://eprint.iacr.org/2023/215.

[7] Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace. Cryptology ePrint Archive, Paper 2023/335, 2023. https://eprint.iacr.org/2023/335.

[8] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [152], 2020. https://sphincs.org/.

[9] Roberto Avanzi, Joppe Bos, Láo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–Kyber: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [152], 2017. https://pq-crystals.org/kyber.

[10] Reza Azarderakhsh, Elena Bakos Lang, David Jao, and Brian Koziel. EdSIDH: Supersingular isogeny Diffie–Hellman key exchange on Edwards curves. In Anupam Chattopadhyay, Chester Rebeiro, and Yuval Yarom, editors, *Security, Privacy, and Applied Cryptography Engineering - 8th International Conference, SPACE 2018, Kanpur, India, December 15-19, 2018, Proceedings*, volume 11348 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2018. https://doi.org/10.1007/978-3-030-05072-6_8.

[11] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Dirmanto Jap, and Dhiman Saha. A survey on fault attacks on symmetric key cryptosystems. *ACM Comput. Surv.*, mar 2022. Just Accepted. https://dl.acm.org/doi/10.1145/3530054.

[12] Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková.

CTIDH: faster constant-time CSIDH. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):351–387, 2021. `https://doi.org/10.46586/tches.v2021.i4.351-387`.

[13] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proc. IEEE*, 94(2):370–382, 2006. `https://doi.org/10.1109/JPROC.2005.862424`.

[14] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021. `https://doi.org/10.1109/SP40001.2021.00008`.

[15] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal collision correlation attack on elliptic curves. In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 553–570. Springer, 2013. `https://doi.org/10.1007/978-3-662-43414-7_28`.

[16] Daniel J. Bernstein. djbsort, 2018. `https://sorting.cr.yp.to`.

[17] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008. `https://doi.org/10.1007/978-3-540-68164-9_26`.

[18] Daniel J. Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters. ECM using Edwards curves. *Math. Comput.*, 82(282):1139–1179, 2013. `https://doi.org/10.1090/S0025-5718-2012-02633-0`.

[19] Daniel J Bernstein, Luca De Feo, Antonin Leroux, and Benjamin Smith. Faster computation of isogenies of large prime degree. *Open Book Series*, 4(1):39–55, 2020. `https://arxiv.org/abs/2003.10118`.

[20] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptogr. Eng.*, 2(2):77–89, 2012. `https://doi.org/10.1007/s13389-012-0027-1`.

[21] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random

strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013. `https://doi.org/10.1145/2508859.2516734`.

[22] Daniel J. Bernstein and Tanja Lange. Analysis and Optimization of Elliptic-Curve Single-Scalar Multiplication. In Gary L. Mullen, Daniel Panario, and Igor E. Shparlinski, editors, *Finite Fields and Applications: Papers from the 8th International Conference*, pages 1–19. Contemporary Mathematics, 461, American Mathematical Society, 2008. `http://eprint.iacr.org/2007/455`.

[23] Daniel J. Bernstein, Tanja Lange, Chloe Martindale, and Lorenz Panny. Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 409–441. Springer, 2019. `https://doi.org/10.1007/978-3-030-17656-3_15`.

[24] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019. `https://doi.org/10.13154/tches.v2019.i3.340-398`.

[25] Thomas Beth, Hans-Joachim Knobloch, and Marcus Otten. Verifiable secret sharing for monotone access structures. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 189–194. ACM, 1993. `https://doi.org/10.1145/168588.168612`.

[26] Luk Bettale, Simon Montoya, and Guénaël Renault. Safe-Error Analysis of Post-Quantum Cryptography Mechanisms - Short Paper -. In *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021*, pages 39–44. IEEE, 2021. `https://doi.org/10.1109/FDTC53659.2021.00015`.

[27] Ward Beullens, Lucas Disson, Robi Pedersen, and Frederik Vercauteren. CSI-RAShi: distributed key generation for CSIDH. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20-22, 2021, Proceedings*, volume 12841 of *Lecture Notes in Computer Science*, pages 257–276. Springer, 2021. `https://doi.org/10.1007/978-3-030-81293-5_14`.

[28] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. CSI-FiSh: efficient isogeny based signatures through class group computations. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 227–247. Springer, 2019. `https://doi.org/10.1007/978-3-030-34578-5_9`.

[29] Patrick Beuth. Verschlüsseln für den Tag X. ZEIT ONLINE, 2015. `https://www.zeit.de/digital/datenschutz/2015-09/post-quanten-kryptografie-tanja-lange-pqcrypto`.

[30] Jean-François Biasse, Annamaria Iezzi, and Michael J. Jacobson Jr. A note on the security of CSIDH. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology - INDOCRYPT 2018 - 19th International Conference on Cryptology in India, New Delhi, India, December 9-12, 2018, Proceedings*, volume 11356 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2018. `https://doi.org/10.1007/978-3-030-05378-9_9`.

[31] G. R. Blakley. Safeguarding cryptographic keys. In Richard E. Merwin, Jacqueline T. Zanca, and Merlin. Smith, editors, *1979 National Computer Conference: June 4–7, 1979, New York, New York*, volume 48 of *AFIPS Conference proceedings*, pages 313–317, pub-AFIPS:adr, 1979. AFIPS Press. `https://doi.org/10.1109/MARK.1979.8817296`.

[32] Johannes Blömer, Ricardo Gomes da Silva, Peter Günther, Juliane Krämer, and Jean-Pierre Seifert. A Practical Second-Order Fault Attack against a Real-World Pairing Implementation. In Assia Tria and Dooho Choi, editors, *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2014, Busan, South Korea, September 23, 2014*, pages 123–136. IEEE Computer Society, 2014. `https://doi.org/10.1109/FDTC.2014.22`.

[33] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997. `https://doi.org/10.1007/3-540-69053-0_4`.

[34] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020. `http://toc.cryptobook.us/`.

[35] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 493–522. Springer, 2020. `https://doi.org/10.1007/978-3-030-45724-2_17`.

[36] Xavier Bonnetain and André Schrottenloher. Quantum Security Analysis of CSIDH and Ordinary Isogeny-based Schemes. Cryptology ePrint Archive, Report 2018/537, 2018. `https://eprint.iacr.org/2018/537`.

[37] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10:113122–113130, 2022. `https://doi.org/10.1109/ACCESS.2022.3217212`.

[38] Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. On the practicality of post-quantum TLS using large-parameter CSIDH. Cryptology ePrint Archive, Paper 2023/793, 2023. `https://eprint.iacr.org/2023/793`.

[39] Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 526–545. Springer, 2020. `https://doi.org/10.1007/978-3-030-65411-5_26`.

[40] Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations Against Fault Injection Attacks. In *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September 13, 2020*, pages 57–65. IEEE, 2020. `https://doi.org/10.1109/FDTC51366.2020.00015`.

[41] Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. LMS vs XMSS: comparison of stateful hash-based signature schemes on ARM Cortex-M4. In Abderrahmane Nitaj and Amr M. Youssef, editors, *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*, volume 12174 of *Lecture Notes in Computer Science*, pages 258–277. Springer, 2020. `https://doi.org/10.1007/978-3-030-51938-4_13`.

[42] Fabio Campos, Juliane Krämer, and Marcel Müller. Safe-error attacks on SIKE and CSIDH. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 11th International Conference, SPACE 2021, Kolkata, India, December 10-13, 2021, Proceedings*, volume 13162 of *Lecture Notes in Computer Science*, pages 104–125. Springer, 2021. `https://doi.org/10.1007/978-3-030-95085-9_6`.

[43] Fabio Campos, Michael Meyer, Krijn Reijnders, and Marc Stöttinger. Patient zero and patient six: Zero-value and correlation attacks on CSIDH and SIKE. In Benjamin Smith and Huapeng Wu, editors, *Selected Areas in Cryptography - SAC 2022 - 29th International Conference, Ontario, Canada, August 24-26, 2022, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2022. `https://eprint.iacr.org/2022/904`.

[44] Fabio Campos, Michael Meyer, Steffen Sanwald, Marc Stöttinger, and Yi Wang. Post-quantum cryptography for ECU security use cases. In *17th escar Europe : embedded security in cars (Konferenzveröffentlichung)*. 2019. `https://doi.org/10.13154/294-6673`.

[45] Fabio Campos and Philipp Muth. On actively secure fine-grained access structures from isogeny assumptions. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 375–398. Springer, 2022. `https://eprint.iacr.org/2021/1109`.

[46] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). *IACR Cryptol. ePrint Arch.*, page 975, 2022. `https://eprint.iacr.org/2022/975`.

[47] Wouter Castryck, Thomas Decru, and Frederik Vercauteren. Radical isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 493–519. Springer, 2020.

[48] Wouter Castryck, Steven Galbraith, and Reza Rezaeian Farashahi. Efficient arithmetic on elliptic curves using a mixed Edwards-Montgomery representation. Cryptology ePrint Archive, Report 2008/218, 2008. `http://eprint.iacr.org/2008/218`.

[49] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 395–427. Springer, 2018. `https://doi.org/10.1007/978-3-030-03332-3_15`.

[50] Wouter Castryck, Lorenz Panny, and Frederik Vercauteren. Rational isogenies from irrational endomorphisms. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 523–548. Springer, 2020. `https://doi.org/10.1007/978-3-030-45724-2_18`.

[51] Daniel Cervantes-Vázquez, Mathilde Chenu, Jesús-Javier Chi-Domínguez, Luca De Feo, Francisco Rodríguez-Henríquez, and Benjamin Smith. Stronger and faster side-channel protections for CSIDH. In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, volume 11774 of *Lecture Notes in Computer Science*, pages 173–193. Springer, 2019. `https://doi.org/10.1007/978-3-030-30530-7_9`.

[52] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002. `https://doi.org/10.1007/3-540-36400-5_3`.

[53] Denis Xavier Charles, Kristin E. Lauter, and Eyal Z. Goren. Cryptographic hash functions from expander graphs. *J. Cryptol.*, 22(1):93–113, 2009.

[54] Jorge Chávez-Saab, Jesús-Javier Chi-Domínguez, Samuel Jaques, and Francisco Rodríguez-Henríquez. The SQALE of CSIDH: sublinear Vélu quantum-resistant isogeny action with low exponents. *Journal of Cryptographic Engineering*, Aug 2021. `https://doi.org/10.1007/s13389-021-00271-w`.

[55] Jesús-Javier Chi-Domínguez and Krijn Reijnders. Fully projective radical isogenies in constant-time. In Steven D. Galbraith, editor, *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, volume 13161 of *Lecture Notes in Computer Science*, pages 73–95. Springer, 2022. `https://doi.org/10.1007/978-3-030-95312-6_4`.

[56] Jesús-Javier Chi-Domínguez and Francisco Rodríguez-Henríquez. Optimal strategies for CSIDH. *Adv. Math. Commun.*, 16(2):383–411, 2022. `https://doi.org/10.3934/amc.2020116`.

[57] Andrew M. Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Math. Cryptol.*, 8(1):1–29, 2014. `https://doi.org/10.1515/jmc-2012-0016`.

[58] Sreeja Chowdhury, Ana Covic, Rabin Yu Acharya, Spencer Dupee, Fatemeh Ganji, and Domenic Forte. Physical security in the post-quantum era: A survey on side-channel analysis, random number generators, and physically unclonable functions. *CoRR*, abs/2005.04344, 2020. `https://arxiv.org/abs/2005.04344`.

[59] Craig Costello. Supersingular Isogeny Key Exchange for Beginners. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 21–50. Springer, 2019. `https://doi.org/10.1007/978-3-030-38471-5_2`.

[60] Craig Costello. B-SIDH: Supersingular Isogeny Diffie–Hellman Using Twisted Torsion. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 440–463. Springer, 2020. `https://doi.org/10.1007/978-3-030-64834-3_15`.

[61] Craig Costello. The case for SIKE: A decade of the supersingular isogeny problem. *IACR Cryptol. ePrint Arch.*, page 543, 2021. `https://eprint.iacr.org/2021/543`.

[62] Craig Costello and Hüseyin Hisil. A simple and compact algorithm for SIDH with arbitrary degree isogenies. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December*

3-7, 2017, Proceedings, Part II, volume 10625 of *Lecture Notes in Computer Science*, pages 303–329. Springer, 2017. https://doi.org/10.1007/978-3-319-70697-9_11.

[63] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie–hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016. https://doi.org/10.1007/978-3-662-53018-4_21.

[64] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie–hellman. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.

[65] Craig Costello, Patrick Longa, and Michael Naehrig. SIDH Library v2.0, 2016. https://github.com/Microsoft/PQCrypto-SIDH/tree/v2.0.

[66] Jean-Marc Couveignes. Hard homogeneous spaces. Cryptology ePrint Archive, Report 2006/291, 2006. https://eprint.iacr.org/2006/291.

[67] Daniele Cozzo and Nigel P. Smart. Sharing the LUOV: threshold post-quantum signatures. In Martin Albrecht, editor, *Cryptography and Coding - 17th IMA International Conference, IMACC 2019, Oxford, UK, December 16-18, 2019, Proceedings*, volume 11929 of *Lecture Notes in Computer Science*, pages 128–153. Springer, 2019. https://doi.org/10.1007/978-3-030-35199-1_7.

[68] Daniele Cozzo and Nigel P. Smart. Sashimi: Cutting up CSI-FiSh secret keys to produce an actively secure distributed signing protocol. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2020. https://doi.org/10.1007/978-3-030-44223-1_10.

[69] Ivan Damgård and Rune Thorbek. Linear integer secret sharing and distributed exponentiation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC*

2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings, volume 3958 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2006. `https://doi.org/10.1007/11745853_6`.

[70] Luca De Feo. Mathematics of isogeny based cryptography. *CoRR*, abs/1711.04062, 2017. `http://arxiv.org/abs/1711.04062`.

[71] Luca De Feo and Steven D. Galbraith. SeaSign: compact isogeny signatures from class group actions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 759–789. Springer, 2019. `https://doi.org/10.1007/978-3-030-17659-4_26`.

[72] Luca De Feo, Jean Kieffer, and Benjamin Smith. Towards practical key exchange from ordinary isogeny graphs. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 365–394. Springer, 2018. `https://doi.org/10.1007/978-3-030-03332-3_14`.

[73] Luca De Feo and Michael Meyer. Threshold schemes from isogeny assumptions. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II*, volume 12111 of *Lecture Notes in Computer Science*, pages 187–212. Springer, 2020. `https://doi.org/10.1007/978-3-030-45388-6_7`.

[74] Luca De Feo, Nadia El Mrabet, Aymeric Genêt, Novak Kaluderovic, Natacha Linard de Guertechin, Simon Pontié, and Élise Tasso. SIKE channels zero-value side-channel attacks on SIKE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):264–289, 2022. `https://doi.org/10.46586/tches.v2022.i3.264-289`.

[75] Victoria de Quehen, Péter Kutas, Chris Leonardi, Chloe Martindale, Lorenz Panny, Christophe Petit, and Katherine E. Stange. Improved torsion-point attacks on SIDH variants. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of

Lecture Notes in Computer Science, pages 432–470. Springer, 2021. https://doi.org/10.1007/978-3-030-84252-9_15.

[76] Bert den Boer. Diffie-hellman is as strong as discrete log for certain primes. In Shafi Goldwasser, editor, *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1988, Proceedings*, volume 403 of *Lecture Notes in Computer Science*, pages 530–539. Springer, 1988. https://doi.org/10.1007/0-387-34799-2_38.

[77] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. https://doi.org/10.1109/TIT.1976.1055638.

[78] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir transformation in the quantum random-oracle model. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 356–383. Springer, 2019. https://doi.org/10.1007/978-3-030-26951-7_13.

[79] Julien Duman, Dominik Hartmann, Eike Kiltz, Sabrina Kunzweiler, Jonas Lehmann, and Doreen Riepel. Group action key encapsulation and non-interactive key exchange in the QROM. *IACR Cryptol. ePrint Arch.*, page 1230, 2022. https://eprint.iacr.org/2022/1230.

[80] Harold Edwards. A normal form for elliptic curves. *Bulletin of the American mathematical society*, 44(3):393–422, 2007. https://doi.org/10.1090/S0273-0979-07-01153-6.

[81] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1857–1874. ACM, 2017. https://doi.org/10.1145/3133956.3134028.

[82] Reza Rezaeian Farashahi and Seyed Gholamhossein Hosseini. Differential addition on twisted edwards curves. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3-5, 2017, Proceedings, Part II*, volume 10343 of *Lecture Notes in Computer Science*, pages 366–378. Springer, 2017. https://doi.org/10.1007/978-3-319-59870-3_21.

[83] Armando Faz-Hernández, Julio César López-Hernández, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie–hellman key exchange protocol. *IEEE Trans. Computers*, 67(11):1622–1636, 2018. https://doi.org/10.1109/TC.2017.2771535.

[84] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93. Springer, 2020. https://doi.org/10.1007/978-3-030-64837-4_3.

[85] Tako Boris Fouotsa. SIDH with masked torsion point images. *IACR Cryptol. ePrint Arch.*, page 1054, 2022. https://eprint.iacr.org/2022/1054.

[86] Tako Boris Fouotsa and Christophe Petit. A new adaptive attack on SIDH. In Steven D. Galbraith, editor, *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, volume 13161 of *Lecture Notes in Computer Science*, pages 322–344. Springer, 2022. https://doi.org/10.1007/978-3-030-95312-6_14.

[87] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *Lecture Notes in Computer Science*, pages 254–271. Springer, 2013. https://doi.org/10.1007/978-3-642-36362-7_17.

[88] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 537–554. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_34.

[89] Steven D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012. https://www.math.auckland.ac.nz/%7Esgal018/crypto-book/crypto-book.html.

[90] Steven D. Galbraith. Authenticated key exchange for SIDH. Cryptology ePrint Archive, Report 2018/266, 2018. `https://eprint.iacr.org/2018/266`.

[91] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 63–91, 2016. `https://doi.org/10.1007/978-3-662-53887-6_3`.

[92] Steven D. Galbraith, Christophe Petit, and Javier Silva. Identification protocols and signature schemes based on supersingular isogeny problems. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2017. `https://doi.org/10.1007/978-3-319-70694-8_1`.

[93] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985. `https://doi.org/10.1109/TIT.1985.1057074`.

[94] Alexandre Gélin and Benjamin Wesolowski. Loop-abort faults on supersingular isogeny cryptosystems. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, volume 10346 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2017. `https://doi.org/10.1007/978-3-319-59879-6_6`.

[95] Aymeric Genêt, Natacha Linard de Guertechin, and Novak Kaluderovic. Full key recovery side-channel attack against ephemeral SIKE on the Cortex-M4. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, volume 12910 of *Lecture Notes in Computer Science*, pages 228–254. Springer, 2021. `https://doi.org/10.1007/978-3-030-89915-8_11`.

[96] Nahid Farhady Ghalaty, Aydin Aysu, and Patrick Schaumont. Analyzing and eliminating the causes of fault sensitivity analysis. In Gerhard P. Fettweis and Wolfgang Nebel, editors, *Design, Automation &*

Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014, pages 1–6. European Design and Automation Association, 2014. https://doi.org/10.7873/DATE.2014.217.

[97] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2012. https://doi.org/10.1007/978-3-642-33481-8_17.

[98] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A "paradoxical'"solution to the signature problem (abstract). In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, page 467. Springer, 1984. https://doi.org/10.1007/3-540-39568-7_37.

[99] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988. https://doi.org/10.1137/0217017.

[100] Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003. https://doi.org/10.1007/3-540-36288-6_15.

[101] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC 1996*, pages 212–219, 1996. https://arxiv.org/abs/quant-ph/9605043.

[102] Javier Herranz and Germán Sáez. Verifiable secret sharing for general access structures, with application to fully distributed proxy signatures. In Rebecca N. Wright, editor, *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*, volume 2742 of *Lecture Notes in Computer Science*, pages 286–302. Springer, 2003. https://doi.org/10.1007/978-3-540-45126-6_21.

[103] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. In Josef Pieprzyk, editor,

Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2008. `https://doi.org/10.1007/978-3-540-89255-7_20`.

[104] Aaron Hutchinson, Jason T. LeGrow, Brian Koziel, and Reza Azarderakhsh. Further optimizations of CSIDH: A systematic approach to efficient strategies, permutations, and bound vectors. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 481–501. Springer, 2020. `https://doi.org/10.1007/978-3-030-57808-4_24`.

[105] Tetsuya Izu and Tsuyoshi Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2003. `https://doi.org/10.1007/3-540-36288-6_17`.

[106] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 632–649. IEEE, 2022. `https://doi.org/10.1109/SP46214.2022.9833713`.

[107] Ján Jančár. The state of tooling for verifying constant-timeness of cryptographic implementations, 2021. `https://neuromancer.sk/article/26`.

[108] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, David Urbanik, Geovandro Pereira, Koray Karabina, and Aaron Hutchinson. SIKE: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [152], 2022. `https://sike.org/`.

[109] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor,

Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011. https://doi.org/10.1007/978-3-642-25405-5_2.

[110] David Jao, Stephen D. Miller, and Ramarathnam Venkatesan. Expander graphs based on GRH with an application to elliptic curve cryptography. *Journal of Number Theory*, 129(6):1491–1504, 2009. https://doi.org/10.1016/j.jnt.2008.11.006.

[111] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 32–61. Springer, 2019. https://doi.org/10.1007/978-3-030-26948-7_2.

[112] Marc Joye and Sung-Ming Yen. The Montgomery Powering Ladder. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2002. https://doi.org/10.1007/3-540-36400-5_22.

[113] David Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet.* Simon and Schuster, 1996.

[114] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[115] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987. https://doi.org/10.1090/S0025-5718-1987-0866109-5.

[116] Neal Koblitz and Alfred Menezes. Critical perspectives on provable security: Fifteen years of "another look" papers. *Adv. Math. Commun.*, 13(4):517–558, 2019. https://doi.org/10.3934/amc.2019034.

[117] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. https://doi.org/10.1109/SP.2019.00002.

[118] Paul C. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. https://doi.org/10.1007/3-540-68697-5_9.

[119] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. https://doi.org/10.1007/3-540-48405-1_25.

[120] Brian Koziel, Reza Azarderakhsh, and David Jao. Side-Channel Attacks on Quantum-Resistant Supersingular Isogeny Diffie–Hellman. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, volume 10719 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2017. https://doi.org/10.1007/978-3-319-72565-9_4.

[121] Adam Langley. ctgrind—checking that functions are constant time with Valgrind, 2010. https://github.com/agl/ctgrind.

[122] Jason LeGrow, David Jao, and Reza Azarderakhsh. Modeling Quantum-Safe Authenticated Key Establishment, and an Isogeny-Based Protocol. Cryptology ePrint Archive, Report 2018/282, 2018. https://eprint.iacr.org/2018/282.

[123] Jason T. LeGrow and Aaron Hutchinson. An Analysis of Fault Attacks on CSIDH. *IACR Cryptol. ePrint Arch.*, page 1006, 2020. https://eprint.iacr.org/2020/1006.

[124] Jason T. LeGrow and Aaron Hutchinson. (Short paper) Analysis of a strong fault attack on static/ephemeral CSIDH. In Toru Nakanishi and Ryo Nojima, editors, *Advances in Information and Computer Security - 16th International Workshop on Security, IWSEC 2021, Virtual Event, September 8-10, 2021, Proceedings*, volume 12835 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 2021. https://doi.org/10.1007/978-3-030-85987-9_12.

[125] Kerstin Lemke-Rust. *Models and algorithms for physical cryptanalysis.* PhD thesis, Ruhr University Bochum, 2007.

[126] John Lennon. Imagine [Song]. In *Imagine*. Universal Music, 1971.

[127] Yang Li, Yu-ichi Hayashi, Arisa Matsubara, Naofumi Homma, Takafumi Aoki, Kazuo Ohta, and Kazuo Sakiyama. Yet another fault-based leakage in non-uniform faulty ciphertexts. In Jean-Luc Danger, Mourad Debbabi, Jean-Yves Marion, Joaquín García-Alfaro, and A. Nur Zincir-Heywood, editors, *Foundations and Practice of Security - 6th International Symposium, FPS 2013, La Rochelle, France, October 21-22, 2013, Revised Selected Papers*, volume 8352 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2013. `https://doi.org/10.1007/978-3-319-05302-8_17`.

[128] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. `https://www.usenix.org/conference/usenixsecurity18/presentation/lipp`.

[129] Patrick Longa. A Note on Post-Quantum Authenticated Key Exchange from Supersingular Isogenies. Cryptology ePrint Archive, Report 2018/267, 2018. `https://eprint.iacr.org/2018/267`.

[130] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2021. `https://doi.org/10.1145/3456629`.

[131] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [152], 2017. `https://pq-crystals.org/dilithium`.

[132] Luciano Maino and Chloe Martindale. An attack on SIDH with arbitrary starting curve. *IACR Cryptol. ePrint Arch.*, page 1026, 2022. `https://eprint.iacr.org/2022/1026`.

[133] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[134] Leandro Marín. Differential elliptic point addition in twisted edwards curves. In Leonard Barolli, Fatos Xhafa, Makoto Takizawa, Tomoya Enokido, and Hui-Huang Hsu, editors, *27th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2013, Barcelona, Spain, March 25-28, 2013*, pages 1337–1342.

IEEE Computer Society, 2013. https://doi.org/10.1109/WAINA.2013.152.

[135] Michael Meyer. *Practical isogeny-based cryptography.* PhD thesis, Julius Maximilians University Würzburg, Germany, 2021. https://doi.org/10.25972/OPUS-24682.

[136] Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*, volume 11505 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2019. https://doi.org/10.1007/978-3-030-25510-7_17.

[137] Michael Meyer and Steffen Reith. A faster way to the CSIDH. In Debrup Chakraborty and Tetsu Iwata, editors, *Progress in Cryptology - INDOCRYPT 2018 - 19th International Conference on Cryptology in India, New Delhi, India, December 9-12, 2018, Proceedings*, volume 11356 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2018. https://doi.org/10.1007/978-3-030-05378-9_8.

[138] Michael Meyer, Steffen Reith, and Fabio Campos. On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic. Cryptology ePrint Archive, Paper 2017/1213, 2017. https://eprint.iacr.org/2017/1213.

[139] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985. https://doi.org/10.1007/3-540-39799-X_31.

[140] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. https://doi.org/10.1090/S0025-5718-1987-0866113-7.

[141] Dustin Moody and Daniel Shumow. Analogues of Vélu's formulas for isogenies on alternate models of elliptic curves. *Math. Comput.*, 85(300):1929–1951, 2016. https://doi.org/10.1090/mcom/3036.

[142] Amir Moradi, Oliver Mischke, and Thomas Eisenbarth. Correlation-enhanced power analysis collision attack. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2010. https://doi.org/10.1007/978-3-642-15031-9_9.

[143] Tomoki Moriya. Masked-degree SIDH. *IACR Cryptol. ePrint Arch.*, page 1019, 2022. https://eprint.iacr.org/2022/1019.

[144] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Secur. Priv.*, 16(5):38–41, 2018. https://doi.org/10.1109/MSP.2018.3761723.

[145] Michael Naehrig and Joost Renes. Dual isogenies and their application to public-key compression for isogeny-based cryptography. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 243–272. Springer, 2019. https://doi.org/10.1007/978-3-030-34621-8_9.

[146] Kohei Nakagawa, Hiroshi Onuki, Atsushi Takayasu, and Tsuyoshi Takagi. L_1-norm ball for CSIDH: Optimal strategy for choosing the secret key space, 2020. https://eprint.iacr.org/2020/181.

[147] Erick Nascimento and Łukasz Chmielewski. Applying horizontal clustering side-channel attacks on embedded ECC implementations. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2017. https://doi.org/10.1007/978-3-319-75208-2_13.

[148] National Institute of Standards and Technology. FIPS186-4: Digital Signature Standard (DSS), 2013. https://doi.org/10.6028/NIST.FIPS.186-4.

[149] National Institute of Standards and Technology. NIST SP 800-56A Rev. 3: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography, 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3.

[150] National Institute of Standards and Technology. NIST SP 800-56B Rev. 2: Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography, 2019. https://doi.org/10.6028/NIST.SP.800-56Br2.

[151] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation,*

San Diego, California, USA, June 10-13, 2007, pages 89–100. ACM, 2007. https://doi.org/10.1145/1250734.1250746.

[152] NIST Computer Security Division. Post-Quantum Cryptography Standardization, 2016. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography.

[153] David Noack, Steffen Sanwald, Marc Stöttinger, Martin Böhner, Norman Lahr, Thorsten Knoll, Steffen Reith, Evangelos Karatsiolis, Georg Land, Juliane Krämer, and Marcel Müller. Industrial use cases and requirements for the deployment of post-quantum cryptography, 2020. https://www.quantumrisc.de/results/quantumrisc-wp1-report.pdf.

[154] Hiroshi Onuki, Yusuke Aikawa, Tsutomu Yamazaki, and Tsuyoshi Takagi. (Short paper) A faster constant-time algorithm of CSIDH keeping two points. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security - 14th International Workshop on Security, IWSEC 2019, Tokyo, Japan, August 28-30, 2019, Proceedings*, volume 11689 of *Lecture Notes in Computer Science*, pages 23–33. Springer, 2019. https://doi.org/10.1007/978-3-030-26834-3_2.

[155] Rémy Oudompheng and Giacomo Pope. A note on reimplementing the Castryck-Decru attack and lessons learned for SageMath. *IACR Cryptol. ePrint Arch.*, page 1283, 2022. https://eprint.iacr.org/2022/1283.

[156] Lorenz Panny. *Cryptography on Isogeny Graphs.* PhD thesis, TU Eindhoven, Mathematics and Computer Science, February 2021. https://research.tue.nl/en/publications/cryptography-on-isogeny-graphs.

[157] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991. https://doi.org/10.1007/3-540-46766-1_9.

[158] Chris Peikert. He Gives C-Sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*,

pages 463–492. Springer, 2020. https://doi.org/10.1007/978-3-030-45724-2_16.

[159] Arnold K Pizer. Ramanujan graphs and Hecke operators. *Bulletin of the American Mathematical Society*, 23(1):127–137, 1990. https://doi.org/10.1090/S0273-0979-1990-15918-X.

[160] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project [152], 2017. https://falcon-sign.info/.

[161] Joost Renes. Computing isogenies between montgomery curves using the action of (0, 0). In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, volume 10786 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2018. https://doi.org/10.1007/978-3-319-79063-3_11.

[162] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983. https://doi.org/10.1145/357980.358017.

[163] Damien Robert. Breaking SIDH in polynomial time. *IACR Cryptol. ePrint Arch.*, page 1038, 2022. https://eprint.iacr.org/2022/1038.

[164] A. Rostovtsev and A. Stolbunov. Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145, 2006. http://eprint.iacr.org/2006/145.

[165] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015. https://doi.org/10.1007/978-3-662-48324-4_25.

[166] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM*

SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020, pages 1461–1480. ACM, 2020. `https://doi.org/10.1145/3372297.3423350`.

[167] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. Supersingular Isogeny Key Encapsulation (SIKE) Round 2 on ARM Cortex-M4. *IEEE Trans. Computers*, 70(10):1705–1718, 2021. `https://doi.org/10.1109/TC.2020.3023045`.

[168] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. `http://doi.acm.org/10.1145/359168.359176`.

[169] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against spectre v1. *IACR Cryptol. ePrint Arch.*, page 1270, 2022. `https://eprint.iacr.org/2022/1270`.

[170] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994. `https://doi.org/10.1109/SFCS.1994.365700`.

[171] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134. IEEE Computer Society, 1994. `https://doi.org/10.1109/SFCS.1994.365700`.

[172] Joseph H. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate texts in mathematics*. Springer, 1986.

[173] S. Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, 2000.

[174] Markus Stadler. Publicly verifiable secret sharing. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 1996. `https://doi.org/10.1007/3-540-68339-9_17`.

[175] Seiichiro Tani. Claw finding algorithms using quantum walk. *Theor. Comput. Sci.*, 410(50):5285–5297, 2009. `https://doi.org/10.1016/j.tcs.2009.08.030`.

[176] Tamir Tassa. Hierarchical threshold secret sharing. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*, pages 473–490. Springer, 2004. `https://doi.org/10.1007/978-3-540-24638-1_26`.

[177] Élise Tasso, Luca De Feo, Nadia El Mrabet, and Simon Pontié. Resistance of Isogeny-Based Cryptographic Implementations to a Fault Attack. In Shivam Bhasin and Fabrizio De Santis, editors, *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, volume 12910 of *Lecture Notes in Computer Science*, pages 255–276. Springer, 2021. `https://doi.org/10.1007/978-3-030-89915-8_12`.

[178] The National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`.

[179] Rune Thorbek. Proactive linear integer secret sharing. *IACR Cryptol. ePrint Arch.*, page 183, 2009. `http://eprint.iacr.org/2009/183`.

[180] Yan Bo Ti. Fault attack on supersingular isogeny cryptosystems. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, volume 10346 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2017. `https://doi.org/10.1007/978-3-319-59879-6_7`.

[181] Giulia Traverso, Denise Demirel, and Johannes Buchmann. Performing computations on hierarchically shared secrets. In Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 141–161. Springer, 2018. `https://doi.org/10.1007/978-3-319-89339-6_9`.

[182] David Urbanik and David Jao. SoK: The problem landscape of SIDH. In Keita Emura, Jae Hong Seo, and Yohei Watanabe, editors, *Proceedings of the 5th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS, Incheon, Republic of Korea, June 4, 2018*, pages 53–60. ACM, 2018. `https://doi.org/10.1145/3197507.3197516`.

[183] J. van Woudenberg and C. O'Flynn. *The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks.* No Starch Press, 2021.

[184] Jacques Vélu. Isogénies entre courbes elliptiques. *Comptes Rendus de l'Académie des Sciences de Paris*, 273:238–241, 1971.

[185] Weijia Wang, Yu Yu, François-Xavier Standaert, Junrong Liu, Zheng Guo, and Dawu Gu. Ridge-based DPA: improvement of differential power analysis for nanoscale chips. *IEEE Trans. Inf. Forensics Secur.*, 13(5):1301–1316, 2018. `https://doi.org/10.1109/TIFS.2017.2787985`.

[186] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2022. `https://www.hertzbleed.com/`.

[187] Lawrence C Washington. *Elliptic curves: number theory and cryptography.* Chapman and Hall/CRC, 2008.

[188] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Trans. Computers*, 49(9):967–970, 2000. `https://doi.org/10.1109/12.869328`.

[189] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In Kwangjo Kim, editor, *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, volume 2288 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2001. `https://doi.org/10.1007/3-540-45861-1_31`.

[190] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. RSA Speedup with Chinese Remainder Theorem Immune against Hardware Fault Cryptanalysis. *IEEE Trans. Computers*, 52(4):461–472, 2003. `https://doi.org/10.1109/TC.2003.1190587`.

[191] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A post-quantum digital signature scheme based on supersingular isogenies. In Aggelos Kiayias, editor, *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, volume 10322 of *Lecture Notes in Computer Science*, pages 163–181. Springer, 2017. `https://doi.org/10.1007/978-3-319-70972-7_9`.

[192] Bilgiday Yuce, Nahid Farhady Ghalaty, Chinmay Deshpande, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection

and Software Fault Response. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*, pages 8:1–8:8. ACM, 2016. `https://doi.org/10.1145/2948618.2948626`.

[193] Fan Zhang, Bolin Yang, Xiaofei Dong, Sylvain Guilley, Zhe Liu, Wei He, Fangguo Zhang, and Kui Ren. Side-Channel Analysis and Countermeasure Design on ARM-Based Quantum-Resistant SIKE. *IEEE Trans. Computers*, 69(11):1681–1693, 2020. `https://doi.org/10.1109/TC.2020.3020407`.

Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University, The Netherlands.[39]

The following research datasets have been produced during this PhD research:

- Chapter 3.1: On hybrid SIDH schemes
  `https://github.com/sopmacF/hybrid-SIDH`
- Chapter 4.1: Efficient constant-time implementation of CSIDH
  `https://github.com/sopmacF/On-Lions-and-Elligators`
- Chapter 4.2: CTIDH: faster constant-time CSIDH
  `https://github.com/sopmacF/CTIDH`
- Chapter 5.1: Protecting CSIDH with Dummy-Operations
  `https://github.com/csidhfi/csidhfi`
- Chapter 5.2: Safe-Error Attacks on SIKE and CSIDH
  `https://github.com/Safe-Error-Attacks-on-SIKE-and-CSIDH/SEAoSaC`
- Chapter 5.3: Zero-Value and Correlation Attacks on SIKE and CSIDH
  `https://github.com/PaZeZeVaAt/simulation`

Further, all datasets produced during this thesis packaged into a single archive are available at `https://doi.org/10.5281/zenodo.6900027`.

[39]ru.nl/icis/research-data-management/, last accessed July 20th, 2022.

Summary

The security of public-key protocols widely deployed today relies on the hardness of the integer factorization problem and the discrete logarithm problem. Due to Shor's algorithm these problems can be efficiently solved by a sufficiently large-scale quantum computer. Thus, quantum computers pose a serious threat to today's digital security.

Among other approaches for building quantum-safe algorithms, *isogeny-based cryptography* is a relatively new approach based on the hardness of finding homomorphisms between elliptic curves.

The focus of this thesis lies on optimizations, secure implementations, and applications of isogeny-based cryptography.

On the constructive side, we first present and evaluate a hybrid SIDH scheme based on Montgomery and twisted Edwards curves. Further, we introduce two approaches for evaluating CSIDH in constant-time. In particular, we present the first complete constant-time implementation of CSIDH and CTIDH, a new key space and a corresponding new algorithm achieving speed records.

On the destructive side, we focus on physical attacks on isogeny-based schemes to understand the security of these schemes against powerful adversaries. Thereby, we present several attacks and possible countermeasures on different isogeny-based schemes and their variants.

Finally, we present an actively secure threshold scheme in the setting of hard homogenous spaces.

Samenvatting

De veiligheid van publieke-sleutelcryptografie, wat tegenwoordig op grote schaal gebruikt wordt, berust op de complexiteit van het factorisatieprobleem en het discrete-logaritmeprobleem. Dankzij Shor's algoritme kunnen deze problemen efficiënt worden opgelost door een kwantumcomputer die groot genoeg is. Kwantumcomputers vormen daarmee een ernstige bedreiging voor de huidige digitale veiligheid. Er bestaan verschillende benaderingen voor het bouwen van kwantumveilige algoritmen. Een relatief nieuwe benadering is isogenie-gebaseerde cryptografie, gebaseerd op de complexiteit van het vinden van homomorfismen tussen elliptische krommen. De focus van dit proefschrift ligt op optimalisaties, veilige implementaties en toepassingen van isogenie-gebaseerde cryptografie. Aan de constructieve kant presenteren en evalueren we eerst een hybride SIDH-schema gebaseerd op Montgomery en *twisted* Edwards-krommen. Verder introduceren we twee benaderingen voor het evalueren van CSIDH in constante tijd. In het bijzonder presenteren we de eerste volledige constantetijdimplementatie van CSIDH en CTIDH, een nieuwe key space en een bijbehorend nieuw algoritme waarmee snelheidsrecords worden gehaald. Aan de destructieve kant richten we ons op fysieke aanvallen op schema's gebaseerd op isogenieën om de veiligheid van deze schema's tegen krachtige aanvallers te begrijpen. Daarbij presenteren we verschillende aanvallen op schema's die op isogenieën zijn gebaseerd en hun varianten, en maatregelen om deze aanvallen te voorkomen. Tenslotte presenteren we een *threshold* schema dat actief veilig is, in de setting van *hard homogenous spaces.*

About the Author

Fabio was born in (the wonderful city of) Rio de Janeiro, Brazil, on February 16, 1975. After finishing the *Segundo Grau* at Colégio Pinheiro Guimarães, Rio de Janeiro, Brazil, he moved to Germany. In 1997, he started studying applied computer science at Fachhochschule Wiesbaden, Germany. He completed his Diploma in 2002. After finishing his diploma studies, he changed to the industry, where he was responsible for the IT and process management departments of two companies for over 15 years. In 2006, he started a master's degree at the RheinMain University of Applied Sciences, Wiesbaden, Germany. His master studies ended in 2010 with a M.Sc. thesis on distributed computing in the fields of number theory supervised by Steffen Reith and Jörn Steuding. In 2018, he started his Ph.D. studies on post-quantum cryptography at the RheinMain University of Applied Sciences under the supervision of Steffen Reith. From 2019, he was an external Ph.D. student at the Radboud University, Nijmegen, The Netherlands, under the supervision of Peter Schwabe and Steffen Reith. In 2021, Fabio was researcher at the Max-Planck-Institute for Security and Privacy in Bochum, Germany. This thesis presents results of works from 2017 to 2022.

List of Publications

Fabio Campos, Jorge Chavez-Saab, Jesús-Javier Chi-Domínguez, Michael Meyer, Krijn Reijnders, Francisco Rodríguez-Henríquez, Peter Schwabe, and Thom Wiggers. On the practicality of post-quantum TLS using large-parameter CSIDH. Cryptology ePrint Archive, Paper 2023/793, 2023. `https://eprint.iacr.org/2023/793` (under submission)

Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace. Cryptology ePrint Archive, Paper 2023/335, 2023. `https://eprint.iacr.org/2023/335` (to be published at TCHES 2023)

Fabio Campos, Michael Meyer, Krijn Reijnders, and Marc Stöttinger. Patient zero and patient six: Zero-value and correlation attacks on CSIDH and SIKE. In Benjamin Smith and Huapeng Wu, editors, *Selected Areas in Cryptography - SAC 2022 - 29th International Conference, Ontario, Canada, August 24-26, 2022, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2022. `https://eprint.iacr.org/2022/904` (to be published at SAC 2022)

Fabio Campos and Philipp Muth. On actively secure fine-grained access structures from isogeny assumptions. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*, volume 13512 of *Lecture Notes in Computer Science*, pages 375–398. Springer, 2022. `https://eprint.iacr.org/2021/1109`

Fabio Campos, Juliane Krämer, and Marcel Müller. Safe-error attacks on SIKE and CSIDH. In Lejla Batina, Stjepan Picek, and Mainack Mondal, editors, *Security, Privacy, and Applied Cryptography Engineering - 11th International Conference, SPACE 2021, Kolkata, India, December 10-13, 2021, Proceedings*, volume 13162 of *Lecture Notes in Computer Science*, pages 104–125. Springer, 2021. `https://doi.org/10.1007/978-3-030-95085-9_6`

Gustavo Banegas, Daniel J. Bernstein, Fabio Campos, Tung Chou, Tanja Lange, Michael Meyer, Benjamin Smith, and Jana Sotáková. CTIDH: faster constant-time CSIDH. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):351–387, 2021. `https://doi.org/10.46586/tches.v2021.i4.351-387`

Fabio Campos, Lars Jellema, Mauk Lemmen, Lars Müller, Daan Sprenkels, and Benoît Viguier. Assembly or optimized C for lightweight cryptography on RISC-V? In Stephan Krenn, Haya Shulman, and Serge Vaudenay, editors, *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings*, volume 12579 of *Lecture Notes in Computer Science*, pages 526–545. Springer, 2020. `https://doi.org/10.1007/978-3-030-65411-5_26`

Fabio Campos, Matthias J. Kannwischer, Michael Meyer, Hiroshi Onuki, and Marc Stöttinger. Trouble at the CSIDH: Protecting CSIDH with Dummy-Operations Against Fault Injection Attacks. In *17th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2020, Milan, Italy, September*

13, 2020, pages 57–65. IEEE, 2020. https://doi.org/10.1109/FDTC51366.2020.00015

Fabio Campos, Tim Kohlstadt, Steffen Reith, and Marc Stöttinger. LMS vs XMSS: comparison of stateful hash-based signature schemes on ARM Cortex-M4. In Abderrahmane Nitaj and Amr M. Youssef, editors, *Progress in Cryptology - AFRICACRYPT 2020 - 12th International Conference on Cryptology in Africa, Cairo, Egypt, July 20-22, 2020, Proceedings*, volume 12174 of *Lecture Notes in Computer Science*, pages 258–277. Springer, 2020. https://doi.org/10.1007/978-3-030-51938-4_13

Michael Meyer, Fabio Campos, and Steffen Reith. On Lions and Elligators: An efficient constant-time implementation of CSIDH. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*, volume 11505 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2019. https://doi.org/10.1007/978-3-030-25510-7_17

Fabio Campos, Michael Meyer, Steffen Sanwald, Marc Stöttinger, and Yi Wang. Post-quantum cryptography for ECU security use cases. In *17th escar Europe : embedded security in cars (Konferenzveröffentlichung)*. 2019. https://doi.org/10.13154/294-6673

Michael Meyer, Steffen Reith, and Fabio Campos. On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic. Cryptology ePrint Archive, Paper 2017/1213, 2017. https://eprint.iacr.org/2017/1213