TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informationstechnik
an der Fakultät für Elektrotechnik und Informationstechnik

Curve Based Cryptography: High-Performance Implementations and Speed Enhancing Methods

Claus Philipp Koppermann

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines **Doktor-Ingenieurs (Dr.-Ing.)** genehmigten Dissertation.

Vorsitzender:		Prof. Dr.-Ing. Dr. rer. nat. Holger Boche
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Georg Sigl
	2.	Prof. Dr. rer. nat. Marian Margraf

Die Dissertation wurde am 17.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 13.05.2019 angenommen.

Abstract

Key exchange algorithms based on public-key cryptography are a crucial component in modern communication systems because they enable two parties to securely derive a shared secret over a public channel. Unfortunately, the popular public-key cryptosystem RSA suffers in speed and requires large key sizes. New arising technologies put high demands on latency and throughput characteristics that catalyze the need for fast key exchange implementations. This thesis deals with high-performance implementations and speed enhancing methods of *curve based cryptography* that is composed of elliptic, hyperelliptic, and isogeny-based curve cryptography.

Elliptic and hyperelliptic curve cryptography are both based upon a fast modular arithmetic that make them appealing for securing communications with high-performance requirements. Many modern curves are defined over so-called Mersenne prime fields that feature an efficient reduction procedure. As modular multiplication is a performance critical operation, a novel speed enhancing architecture is introduced for multiplying two elements in a Mersenne prime field. Using this modular multiplier, two highly-optimized key exchange implementations are evaluated on an FPGA based on two different types of curves: Curve25519, a popular elliptic curve, and Gaudry and Schost's Kummer surface of a genus-2 curve, a novel hyperelliptic variant. Both implementations are analyzed and compared regarding their performance and implementation security. The Curve25519 implementation is optimized for low-latency applications and uses randomized projective coordinates to thwart specific side-channel attacks. Though, the performance results of our high-speed Kummer variant outperform all previous prime field curve implementations in terms of latency and throughput. The implemented Kummer architecture smartly interleaves two scalar multiplications at a time, which can be used to double the throughput or enable an inherent countermeasure against fault attacks. Compared to elliptic curve cryptography, the hyperelliptic variant achieves improved area and performance results due to a smaller field size.

It is well known that the continuous progress in the development of a quantum computer threatens the secure application of elliptic and hyper-elliptic curve cryptography. Combined with the advent of the *Internet of Things*, thousands of interconnected nodes that process sensitive information are threatened. Therefore, the applicability and implementation security of the quantum-secure supersingular isogeny Diffie-Hellman (SIDH) key exchange is examined for the embedded scenario. SIDH attracted considerable interest in the research community due to its small key sizes when compared to other post-quantum primitives. While several works already presented speed-optimized FPGA implementations, it remains unclear whether it is suitable for resource-constrained devices. Hence, we provide a software implementation of SIDH over a 751-bit wide extension field, which is considered to provide a quantum-security level of at least 128-bit. Our software implementation is assembly optimized and trimmed towards maximum speed targeting a 32-bit ARM Cortex-M4 and a 16-bit TI MSP430 architecture. However, an ephemeral key exchange still requires more than 18 seconds for the Cortex-M4 and more than 11 minutes for the MSP430. With respect to the chosen parameters, this shows that SIDH is unsuitable for most real-life applications when implemented on small embedded devices.

Kurzfassung

Schlüsselaustauschprotokolle basierend auf Public-Key Kryptographie sind essentiell in modernen Kommunikationsnetzwerken, weil sie zwei Teilnehmern erlauben sich über einen öffentlichen Kanal auf ein gemeinsames Geheimnis zu einigen. Jedoch ist das bekannteste Public-Key Kryptosystem RSA aufwendig in seiner Berechnung und benötigt große Schlüssel. Insbesondere neue Technologien stellen hohe Performance Anforderungen, was die Notwendigkeit von schnellen Schlüsselaustausch Implementierungen katalysiert. Diese Arbeit beschäftigt sich mit hochoptimierten Implementierungen und beschleunigenden Methoden für *kurvenbasierte Kryptographie*, welche sich in elliptische, hyperelliptische und isogeniebasierte Kurven Kryptographie untergliedert.

Elliptische und hyperelliptische Kurven Kryptographie zeichnet sich durch eine schnelle modulare Arithmetik aus, die sie besonders interessant für Implementierungen mit hohen Performance Anforderungen macht. Viele moderne Kurven sind über so genannte Mersenne Primzahl Körper definiert, die über eine effiziente Methode zur modularen Reduktion verfügen. Da die modulare Multiplikation eine zeitkritische Operation ist, wird zunächst eine neue Hardware Architektur für die modulare Multiplikation in Mersenne Primzahl Körpern vorgestellt. Unter Verwendung dieses Multiplizierers werden zwei Schlüsselaustausch Implementierungen für einen FPGA beschrieben, die auf verschiedenen Kurventypen basieren: Curve25519, eine bekannte elliptische Kurve, und Gaudy und Schosts kummersche Fläche einer Kurve vom Geschlecht 2, eine neue hyperelliptische Variante. Beide Implementierungen werden hinsichtlich ihrer Performance und Implementierungssicherheit analysiert und verglichen. Die Curve25519 Implementierung ist für Anwendungen mit niedrigen Latenz Anforderungen optimiert und verwendet randomisierte projektive Koordinaten um bestimmte Seitenkanal Angriffe zu verhindern. Allerdings übertrifft die Kummer Variante alle Performance Ergebnisse früherer Kurven Implementierungen über Primzahl Körper. Darüber hinaus kombiniert die Kummer Architektur zwei skalare Multiplikationen was entweder den Durchsatz verdoppelt oder eine

Gegenmaßnahme für Fehlerangriffe ermöglicht. Aufgrund eines kleineren Körpers, erreicht die hyperelliptische Variante bessere Flächen und Performance Ergebnisse.

Der kontinuierliche Fortschritt in der Entwicklung des Quantencomputers bedroht die sichere Verwendung von elliptischer und hyperelliptischer Kurven Kryptographie. Unter Berücksichtigung aktueller Trends wie beispielsweise dem der *Internet der Dinge*, sind in Zukunft potentiell tausende verbundene Knoten bedroht. Daher wird die Anwendbarkeit und Implementierungssicherheit des quantensicheren supersingulären Isogenie Diffie-Hellman (SIDH) Schlüsselaustausch für eingebettete Systeme untersucht. SIDH ist von besonderem Interesse, da es verglichen zu anderen Post-Quanten Verfahren relativ kleine Schlüssel verwendet. Während diverse Arbeiten bereits geschwindigkeitsoptimierte Hardware Architekturen vorgestellt haben, bleibt es bis heute unklar wie sich eine Anwendung auf kleinen ressourcenbeschränkten Geräten darstellt. Um diese These zu überprüfen wird eine Assembler-optimierte SIDH Implementierung vorgestellt und deren Performance auf einem 32-Bit ARM Cortex-M4 und auf einem 16-Bit TI MSP430 evaluiert. Die vorgestellte Software realisiert SIDH über ein 751-bit großen Erweiterungskörper mit dem ein Quanten Sicherheitslevel von mindestens 128-bit erreicht wird. Ein ephemeral Schlüsselaustausch benötigt auf einem Cortex-M4 mehr als 18 Sekunden und auf einem MSP430 mehr als 11 Minuten. Mit Bezug auf die gewählten Parameter, zeigt dies dass die Laufzeit von SIDH auf einem eingebetteten Controller für reale Anwendungen noch zu lange ist.

Acknowledgements

Firstly, I would like to express my sincere gratitude to Prof. Dr.-Ing. Georg Sigl for giving me the chance to pursue my Ph.D. at the Technische Universität München (TUM) and Prof. Dr. rer. nat. Marian Margraf for being my second examiner.

My special thanks goes to Dr.-Ing. Johann Heyszl, who provided me the opportunity to join the Hardware Security department at Fraunhofer AISEC, and his valuable scientific guidance.

I especially thank my former supervisor Dr.-Ing. Fabrizio De Santis for introducing me to cryptography and contributing to my research.

I thank my colleagues at Fraunhofer AISEC: Robert Hesselbarth, Stefan Hristozov, Dr. rer. nat. Katja Miller, Carsten Rolfes, Marc Schink, Bodo Selmke, Robert Specht, Martin Striegel, Florian Unterstein, Andreas Zankl, and in particular Nisha Jacob for their feedback, cooperation, and friendship.

I thank my students who supported my research in various ways: Abhijith Chikrapla Danappa, Zohaib Khan, and Eduard Pop.

Last but not least, I would like to thank my partner, family, and close friends for supporting me throughout the writing process of this thesis and my life in general.

Contents

Nomenclature

Abbreviations

ASIC	Application-specific integrated circuit
CC	Cycle count
DH	Diffie-Hellman
DLP	Disrete logarithm problem
DPA	Differential power analysis
DSP	Digital signal processing
ECC	Elliptic curve cryptography
ECDH	Elliptic curve Diffie-Hellman
ECDLP	Elliptic curve discrete logarithm problem
EM	Electromagnetic
FA	Full adder
HA	Half adder
HECC	Hyperelliptic curve cryptography
IoT	Internet of Things
LUT	Lookup table
NIST	National Institute of Standardization
PQC	Post-quantum cryptography
RAM	Random-access memory

RCA	Ripple-carry adder
ROM	Read-only memory
RSA	Rivest Shamir Adleman
SIDH	Supersingular isogeny Diffie-Hellman
SoC	System on chip
TP	Throughput
VLSI	Very-large-scale integration

Mathematical symbols

$[k]P$	Scalar multiplication
∞	Point-at-infinity
$\kappa(P)$	Image of P on $\mathcal{K}$
$\langle P\rangle$	Linear combination of point P
$\|k\|$	Bit length of positive integer k
$\mathbb{F}$	Finite field
$\mathbb{F}_p$	Prime field
$\mathbb{P}$	Projective space
$\mathcal{C}$	General hyperelliptic curve
$\mathcal{H}$	Hadamard transform
$\mathcal{J}_{\mathcal{C}}$	Jacobian associated with curve $\mathcal{C}$
$\mathcal{K}_{\mathcal{C}}$	Kummer surface associated with curve $\mathcal{C}$
$\mathcal{O}$	Identity element
$\overline{P}$ or $-P$	Point opposite to P
ϕ	Isogeny
D	Divisor
$D_1 \oplus D_2$	Addition of two divisors D_1, D_2

E	Elliptic curve
$E[\ell]$	ℓ-torsion subgroup of elliptic curve E
$G \cong H$	Group G is isomorphic to group H
$G \times H$	Direct sum of two groups G, H
K	Field
M_p	Mersenne prime
N	Multiplier width
O	Big O notation
P	Point on elliptic curve or partial-product
p	Prime number
$x(P)$	x-coordinate of point P
A	Modular addition
I	Modular inversion
M	Modular multiplication
$\mathrm{M_c}$	Constant modular multiplication
S	Modular Squaring
Z	Modular subtraction

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In 1976, Diffie and Hellman [6] proposed the Diffie-Hellman (DH) key exchange as the first representative of the revolutionary public-key cryptography. With the advent of public-key cryptography, it became possible to exchange encrypted and authenticated messages without requiring a shared secret. Traditionally, the communication partners were needed to exchange a shared secret via some secure physical channel such as a trusted courier. In public-key cryptography each entity possesses a key pair consisting of a widely disseminated public key and a secret private key. Public keys can be used to validate a signature of incoming messages or encrypt outgoing messages, whereas private keys can be used to sign outgoing messages or decrypt incoming messages. Among signing and encrypting messages based on public-key cryptography, two parties can use the DH key exchange to negotiate a shared secret over a public channel i.e. an adversary that can eavesdrop the channel is unable to determine the shared secret. However, due its low efficiency compared to symmetric schemes, public-key cryptography can be impractical for applications with high performance requirements. To benefit from the speed of symmetric cryptography while renouncing the necessity of a preshared secret, so-called hybrid schemes are used where the communication is initiated with a DH key exchange followed by the usage of symmetric schemes for encrypting and authenticating messages.

Even though hybrid schemes improve the latency and throughput of encrypting and authenticating messages, the DH key exchange itself might still be problematic for applications with strict speed boundaries. In some cases, a high latency only causes a bad user experience, however, in safety critical systems, such as modern car systems, a fast key exchange can be required for guaranteeing a safe operation. Throughput is crucial for systems that need to exchange keys with thousands of devices per second. For example, a network server has to serve a large amount of requests leading to high throughput requirements because less servers can mean higher profit. Even for small embedded devices high-speed cryptography can be crucial as reduced run-time can lead to less energy consumption and hence in a longer lasting battery. To cope with those requirements, intensive research was conducted on high-speed public-key cryptography. High-speed cryptography can be separated in the implementation of cryptographic primitives that are optimized towards speed and in the design of high-speed capable cryptographic primitives. In this work, the former problem is addressed.

Until today, RSA [7] is the most well known public-key cryptosystem, though it is unsuitable for high-speed applications due to its high computational complexity. Instead, elliptic and hyperelliptic curve cryptography represent a valid alternative since they feature small field and key sizes. Both, elliptic and hyperelliptic curve cryptography can be used to construct a DH key exchange. While hyperelliptic curve cryptography is recently gaining in interest in the research community, elliptic curve cryptography can be found already today in a variety of applications such as WhatsApp or Tor. The continuous progress in the development of a quantum computer, threatens classic public-key cryptography such as elliptic and hyperelliptic curve cryptography. Yet, an isogeny-based approach i.e. a quantum-secure so-called supersingular Isogeny DH key exchange, which shares some similarities with elliptic curve cryptography, is attracting considerable interest due to its small key sizes. In this thesis, speed-enhancing methods and high-speed implementations of *curve based cryptography* i.e. elliptic, hyperelliptic, isogeny-based cryptography for the application on hardware accelerators and tiny embedded devices are presented.

1.1 The Diffie-Hellman Key Exchange

Suppose that Alice and Bob want to negotiate a secret key k_{AB}. Let $\mathbb{G}$ denote a finite cyclic group, q its order, and g a generator of $\mathbb{G}$, and let the exponentiation operation denote a repeated group operation that is used as a trapdoor function. Then the DH key exchange is described as follows:

1. Alice and Bob agree on $\mathbb{G}$ and g.
2. Alice chooses $k_A \leftarrow \mathbb{Z}_q$ uniformly at random, and computes $h_1 := g^{k_A}$. Then Alice transmits h_1 to Bob.
3. Bob receives h_1. He chooses $k_B \leftarrow \mathbb{Z}_q$ uniformly at random and computes $h_2 := g^{k_B}$. Bob sends h_2 to Alice and outputs the key $k_{AB} := h_1^{k_B} = g^{k_A k_B}$.

4. Alice receives h_2 and outputs the key $k_{AB} := h_2^{k_A} = g^{k_B k_A}$.

Intuitively, a key exchange is considered secure if the key output is unknown to an eavesdropping adversary. Therefore, a necessary requirement is that an adversary is unable to inverse the exponentiation $h_1 := g^{k_A}$ or $h_2 := g^{k_B}$. In other words, an adversary would need to compute $k_A = \log_g h_1$ or $k_B = \log_g h_2$. In case of exponentiation in a cyclic group, this is considered to be a hard problem for classical computers and is known as the discrete logarithm problem (DLP). Note that the hardness of the DLP is only a minimal requirement but not a sufficient one. As the shared secret is often used as an input key for further encryption algorithms, it shall be also indistinguishable from a completely random key of the same length. This assumption is much stronger, but truly holds for the DH key exchange protocols as shown by Boneh [8], which is considered as the *decisional Diffie-Hellman assumption.*

While being secure against passive adversaries, DH key exchange is insecure in the presence of active adversaries. For example, DH key exchange is vulnerable to the *man-in-the-middle attack*, where an adversary exploits the fact that neither Alice nor Bob can proof the authenticity of incoming messages. First, the attacker intercepts the communication between Alice and Bob. Second, the adversary impersonates Bob to exchange a key with Alice and third, the adversary impersonates Alice to exchange a key with Bob. Appropriate methods to authenticate the communication prevent a man-in-the-middle attack. Therefore, DH key exchange is rare in its basic form, however, it constitutes the nucleus of further protected key exchange protocols. All cyclic groups in which the group operation features an efficient trapdoor function can be used to construct an efficient and secure DH key exchange. For example, the elliptic curve DH protocol is a variant which constructs a group over elliptic curves that enables an efficient arithmetic while providing small keys. Similarly, DH key exchange can be constructed for hyperelliptic curves and isogeny-based approaches. The motivation and underlying mathematical problems for elliptic, hyperelliptic, and isogeny-based cryptography are discussed in the following sections.

1.1.1 Elliptic Curve Cryptography

In 1985, Koblitz [9] and Miller [10] independently discovered elliptic curve cryptography (ECC). The security of a public-key system using elliptic curves is based on the difficulty of computing the discrete logarithm in the group of points on an elliptic curve defined over a finite field. An abelian group is formed by all points on the elliptic curve together with the point at infinity under the addition law, which is obtained by the *chord-and-tangent rule* (see

Chapter 3 for more information). A point can be multiplied with a scalar by using an algorithm such as the Montgomery ladder [4], which repetitively performs point addition and point doubling operations. Finding this scalar with known input and output point forms the elliptic curve discrete logarithm problem (ECDLP), which is currently believed to be asymptotically harder than the factorization of integers or the computation of discrete logarithms in the multiplicative group of a finite field [11]. Compared to RSA and DLP, ECC uses shorter keys while providing the same security level because of the increased hardness of the ECDLP. As a rule of thumb, the key size is about half the number of bits that represent the underlying finite field.

Over the years, many elliptic curves have been standardized by governmental institutions like the American National Institute of Standards and Technology (NIST) or the German Bundesamt für Sicherheit in der Informationstechnik. However, after the Snowden's leak, a growing interest around new elliptic curves has been manifested by the whole cryptographic community. In Chapter 4, we particularly focus on Curve25519, which is a 128-bit secure elliptic curve introduced by Bernstein [12] in 2006. Curve25519 is designed in an elegant and transparent way while offering high-performance characteristics. Therefore, Curve25519 has received wide attention in the past years with various hardware and software implementations being published that set new speed records.

1.1.2 Hyperelliptic Curve Cryptography

In 1989, Koblitz [13] first mentioned the application of hyperelliptic curve cryptography (HECC). For example, the so-called Jacobian variety of a hyperelliptic curve possesses a group structure that can be used to realize cryptographic algorithms such as DH key exchange and digital signatures (see Chapter 5 for a more detailed discussion on hyperelliptic curves). Unfortunately, group operations on the Jacobian have higher complexity than those on elliptic curves (genus-1 curves). However, using the group operation on the Kummer surface of the Jacobian in place of the Jacobian itself, leads to a decrease of the number of field operations per group operation [14]. The Kummer surface is a 2-to-1 point mapping and can be compared to the x-coordinate-only representation of elliptic curves. Table 1.1 shows the number of field operations for a point addition and a point doubling operation used in DH key exchange for a genus-1 Montgomery curve and a Kummer surface associated to a genus-2 curve. It can be noted that the genus-2 curve requires 1.4-times more multiplications, 3-times more squarings, and 4-times more additions and subtractions than the genus-1 curve. However, the Kummer surface operates on finite fields of half the size than

Table 1.1: Required field operations for point addition and point doubling: multiplication (M), squaring (S), constant multiplication (M_c), addition (A), and subtraction (Z).

Genus	**Reference**	**Field size**	**M**	**S**	**M_c**	**A**	**Z**
1	Curve25519 [15]	255-bit	5	4	1	4	4
2	Kummer [16]	127-bit	7	12	12	16	16

those of elliptic curves while supporting the same security level. This reduced field size can lead to performance benefits and lower area utilization. In 2006, Bernstein and Lange [17] showed in a cost analysis for software that a genus-2 based implementation is potentially 1.5-times faster than a comparable elliptic curve based implementation. At that time, however, a secure Kummer surface of a genus-2 curve was not found yet. Since genus-2 point counting is computationally expensive, it took further six years until Gaudry and Schost [18] presented a twist-secure Kummer surface targeting a 128-bit security level. Using this Kummer surface, Bos et al. [19] were the first to publish a high-speed DH implementation on high-end CPUs proving the earlier cost analysis in [17]. Other software implementations [16, 20] on different architectures were published in following years. While these software implementations already showed the performance advantages of genus-2 curves, the design of efficient hardware is a fundamentally different task.

1.1.3 Isogeny-Based Cryptography

It is well known that future large-scale quantum computers can efficiently compute Shor's algorithm [21], and thus threaten public-key cryptosystems that rely on the ECDLP, DLP, or RSA. Even though full-fledged quantum computers are yet to arrive, today's recorded encrypted communication could be broken with a quantum computer years later. In the past few years, this led to intensive research and a large amount of published papers dealing with post-quantum cryptography (PQC) i.e. cryptographic algorithms that are considered to be secure against an attack by a quantum computer. NIST [22] published a report on PQC providing an overview of existing algorithms including an announcement for standardization. In this report, NIST distinguishes between five approaches: lattice-based cryptography, code-based cryptography, multivariate polynomial cryptography, hash-based signatures, and *other* which include isogeny-based cryptography. When analyzing different PQC approaches, it becomes apparent that most of them require large private and public keys. Large key sizes imply at least two problems for smaller embedded devices: First, since the transmission of data requires the majority of the energy budget, the size of the public parameters including the public key must be kept small. Second, small embedded devices often possess less than ten kilobytes of memory. Therefore, PQC algorithms that feature large key pairs, as for example the McEliece cryptosystem [23] that needs about 220 kB for a single public key at a 128-bit quantum security level, are impractical on such devices. With public keys as small as 330 bytes [24], the quantum-secure supersingular isogeny Diffie-Hellman (SIDH) key exchange [5] is a promising candidate to secure the communication on embedded devices.

SIDH is based on elliptic curves and shares similarities with traditional ECC; however, the underlying number-theoretic problem is the isogeny-graph problem. An isogeny is an algebraic map between two elliptic curves, which are defined over a finite field. The point multiplication of a point with some scalar, which is well known in traditional ECC, can be seen as a special case of an isogeny for identical curves. Finding the isogeny between the known domain and co-domain (in case of distinct elliptic curves) constitutes the isogeny-graph problem, which is an instance of the so-called claw problem [5]. This isogeny-graph consists of vertices representing isomorphism classes of elliptic curves that are connected by edges representing isogenies. Alice and Bob start from the vertex that is the public curve and traverse this graph via a seemingly random walk. Ultimately, they end up on two curves sharing some value that is used as the shared secret. While SIDH is a still growing research topic, it remains unclear how it performs in microcontrollers that are typically used in the IoT context.

1.2 Contribution

In the previous sections, different approaches in curve based cryptography were presented that can be applied to DH key exchange. Compared to traditional approaches in cryptography, curve based cryptography features relatively small keys. This thesis deals with methods and implementations that aid future research in obtaining high-speed key exchange implementations using curve based cryptography. The main contributions are summarized below:

Novel design of a modular multiplier using Mersenne primes. Curve based cryptography operates on finite fields. Thus, approaches in curve based cryptography can differ on an algorithmic level, but all require the implementation of field operations. Modular multiplication is a time critical application due to its frequent operation and increased complexity when compared to addition or subtraction. As a first contribution, a high-speed modular multiplier [25], which smartly combines the summation of single digit-products with the reduction step, is presented. This reduces the computational complexity and increases the maximum clock frequency. The multiplier sets a strong foundation for following high-speed implementations.

Low latency X25519 implementation on FPGA. Curve25519 and its corresponding key exchange X25519 is widely adopted in commercial solutions such as WhatsApp and Tor. Therefore, a latency optimized X25519 implementation on FGPA [26, 27] is presented. Moreover, the design is protected against differential power analysis (DPA) using randomized projective coordinates as an efficient countermeasure. The performance results show that X25519 enables a fast but also an area demanding implementation.

High-speed key exchange based on a hyperelliptic curve. As shown in previous software implementations, a high-speed DH key exchange can be implemented using the Kummer surface of a genus-2 curve. The reduced field size, which is half the size than those of elliptic curves while supporting the same security level, allows for fast implementations. Therefore, the first FPGA implementation of a DH key exchange based upon the Kummer surface of hyperelliptic curve [28] is reported, which shows outstanding latency and throughput results. The implementation includes a novel technique that interleaves two scalar multiplications at a time to effectively double the throughput. The same technique can also be used as redundancy countermeasure against fault attacks.

Evaluating SIDH on embedded devices. SIDH is a quantum secure key exchange that is characterized by small keys. Therefore, it is seems to be appealing for securing embedded devices with constrained resources. This hypothesis is evaluated by presenting a speed optimized SIDH software implementation for two popular microcontroller architectures [29]. The results indicate that SIDH over a 751-bit wide extension field is impractical on embedded devices due its long computation time. Moreover, the implementation security of SIDH is analyzed by measuring its electromagnetic radiation during critical operations.

The details above just briefly highlighted the main contributions of this thesis. The list of publications that correspond to those contributions can be found below:

[25] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Automatic generation of high-performance modular multipliers for arbitrary Mersenne primes on FPGAs. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, pages 35–40, 2017

[26] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. X25519 hardware implementation for low-latency applications. In *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*, pages 99–106, 2016

[27] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Low-latency X25519 hardware implementation: Breaking the 100 microseconds barrier. *Microprocessors and Microsystems - Embedded Hardware Design*, 52:491–497, 2017

[28] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Fast FPGA implementations of Diffie-Hellman on the Kummer surface of a genus-2 curve. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):1–17, 2018

1.3 Outline

In Chapter 2, a novel design of a modular multiplier for the application on ASIC and FPGA is presented. In Chapter 3, the reader is provided with background information on ECC. Chapter 4 discusses the latency-optimized implementation of X25519 on FPGA. Chapter 5 details the theoretical foundations of HECC. A Kummer surface based key exchange using a genus-2 curve is presented in Chapter 6. In Chapter 7, background information on isogeny-based cryptography is presented and in Chapter 8, an implementation of SIDH on embedded devices is discussed. Finally, Chapter 9 concludes.

Chapter 2

Multiplication in Mersenne Prime Fields

The performance of curve based cryptography strongly depends on the implementation of the underlying field operations i.e. modular addition, subtraction, multiplication, squaring, and inversion. With regard to high-speed applications, the implementation of the modular multiplication should be optimized thoroughly due to its frequent usage and computational complexity. We focus on multiplication in prime fields and in particular on multiplication in Mersenne prime fields. Curves defined over Mersenne prime fields gained in importance due to Crandall's [30] efficient reduction procedure. Prominent examples are Microsoft's Four$\mathbb{Q}$ [31] and the Kummer surface based key exchange [20] for elliptic and hyperelliptic curve cryptography, respectively. We demonstrate how an efficient architecture for the multiplication in Mersenne prime fields can be designed for ASIC as well as FPGA designs. Parts of this chapter have been published in [25].

Outline. In Section 2.1, the preliminaries of multiplication in prime fields are described. Section 2.2 discusses common architectures for hardware multipliers. In Section 2.3, it is shown how Crandall's reduction technique can be combined with long integer multiplication. Section 2.4 formulates the basic assumptions and the problem statement for deriving efficient multipliers on FPGAs. In Section 2.5, related work that deals with fast multiplication on FPGAs is summarized. Section 2.6 provides a description of the combined modular multiplication and the algorithms for automatically generating the corresponding hardware architectures. Section 2.7 presents the implementation results while Section 2.8 concludes.

2.1 Preliminaries

Modular multiplication in a prime field $\mathbb{F}_p$, also known as prime field multiplication, is the mathematical operation

$$C \equiv A \cdot B \pmod{p}$$

where $A, B \in \mathbb{F}_p$, and p is a prime number. Modular multiplication can be classified in classic and interleaved techniques: For classic modular multiplication, a standard long integer multiplication algorithm is used, as for example the well-known schoolbook method, which is subsequently followed by the reduction procedure. The designer has a broad choice between multiplication algorithms due to extensive research in the past decades. However, as the size of the product is twice the size of the operands, time and area requirements can increase. For interleaved modular multiplication, the multiplication and the calculation of the remainder are interleaved. Intermediate products are similar sized as the reduced products, which is advantageous in terms of area efficiency. Montgomery multiplication [32] is a popular representative of interleaved modular multiplication.

For Mersenne primes, Crandall's reduction procedure [30], which operates on the product of the multiplication, can be applied. Even though our design combines multiplication and reduction, the presented technique builds upon standard schoolbook multiplication. Therefore, we first describe standard multiplication architectures which enables us to introduce a common notation and determines criteria for performance evaluation. The combination of multiplication and Crandall's fast reduction procedure represents an interleaved modular multiplication as it shrinks the intermediate result to the size of the operands.

The contribution of this chapter is twofold: First, we present how Crandall's reduction technique can be combined with long integer multiplication for hardware designs in general such that only standard digital logic blocks are used. The corresponding modular multiplier is constructed by embedding Crandall's reduction technique inside a Wallace tree multiplier. A Wallace tree multiplier is a speed optimized design of a hardware multiplier. It is shown that our approach can execute a multiplication in a Mersenne prime field as fast as a single long integer multiplication using a Wallace tree multiplier. Second, the generalized approach is mapped and optimized for FPGA specific technology i.e. base the multiplication on smaller embedded multipliers that are contained in DSP slices. A formalized approach is presented and algorithms are provided that automatically generate high-performance modular multipliers for arbitrary Mersenne primes from any small-sized (po-

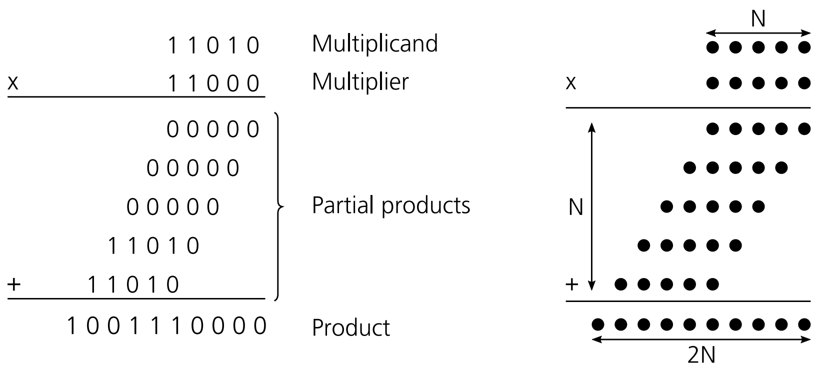


Figure 2.1: Left: Schoolbook multiplication as 5-bit binary multiplication. Right: Corresponding dot-representation.

tentially asymmetric) multipliers i.e. not being limited to current DSP technologies. These algorithms were implemented in Python and placed in the public domain[1]. The proposed design can perform multiplication and reduction with almost the same latency as previous works that only perform multiplication, yet achieving a throughput with a 1.36-factor improvement.

2.2 Hardware Multipliers

Most multiplier architectures follow a similar procedure: compute a set of partial-products and subsequently accumulate those using an adder circuit. Figure 2.1 exemplary depicts a 5-bit binary multiplication and its corresponding dot-notation where each dot is a placeholder for a single bit, which can be zero or one. A partial-product is formed by a horizontal row of dots. The illustrated multiplication algorithm is the popular schoolbook multiplication but operates on base-2 instead of base-10 integers. In each step the algorithm selects a multiplier bit from right to left and computes a logically-and with the 5-bit multiplicand. Depending on the current bit position of the multiplier, the partial-product is shifted to the left. Finally, all computed partial-products are accumulated to the product.

Array Multiplier

A multiplier can be implemented in a serial, parallel, or hybrid fashion, which depends on the performance and area requirements. An exemplary hybrid

[1]`https://github.com/Fraunhofer-AISEC/mod-mul-mersenne`

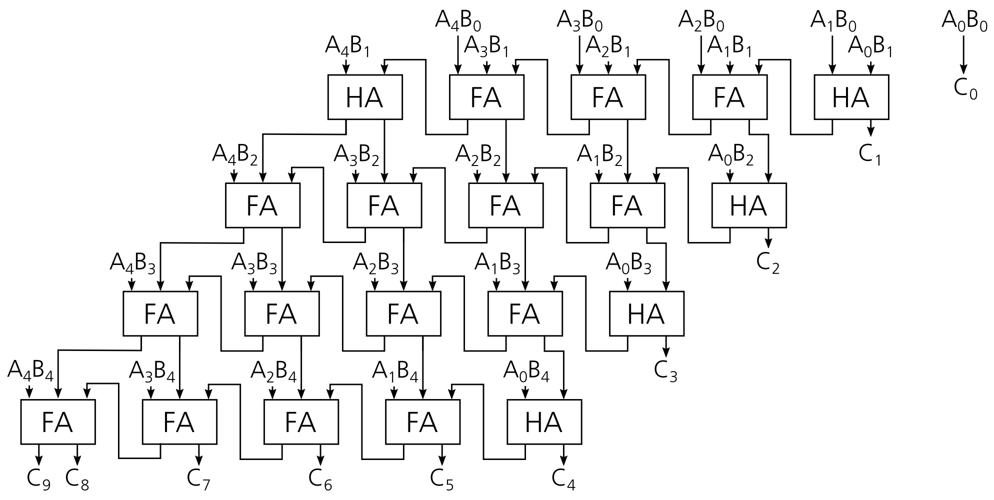


Figure 2.2: Architecture of a hybrid 5-bit array multiplier.

array multiplier is shown in Figure 2.2. This architecture is the direct translation of the schoolbook algorithm from Figure 2.1 to hardware. Again, a partial-product is generated by the multiplication of the multiplicand with a single multiplier bit by applying logical-and bitwise. The partial-products are shifted according to their bit orders and then accumulated using a standard carry propagate adder which is composed of full adders (FAs) and half adders (HAs). We determine the delay of the array multiplier as follows: The logical-and for the single bit multiplication has a delay of $O(1)$. Based on Figure 2.2, the longest path of adders is given by $O(N)$, in which each adder has a delay of $O(1)$. Thus, the overall delay of an array multiplier is $O(N)$. Here, delay refers to the time complexity, which describes the required amount of time to run an algorithm. Compared to full parallel multiplier designs, the array multiplier suffers in speed; on the other hand, it features a regular structure, which makes it appealing for VLSI.

Wallace Tree Multiplier

A multiplier featuring a fully parallel adder tree, i.e. accumulating the partial-products in parallel, is well suited for applications with high-speed requirements. In 1964, Wallace [33] described such an efficient adder tree, which is constructed by three steps:

1. Generate the partial-products by combining the multiplicand with the multiplier using a logical-and bitwise.

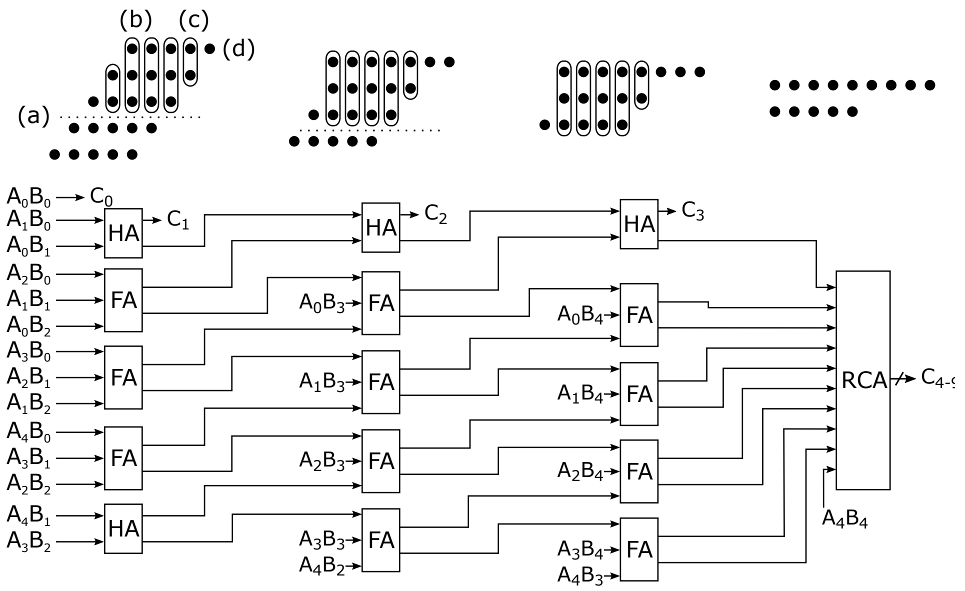


Figure 2.3: Constructing a 5-bit Wallace tree multiplier. Explanation of (a), (b), (c), and (d) is found below.

2. Reduce the number of partial-products by partitioning them in layers in which the 1-bit products are combined using full and half adders (see explanation below). Continue until two partial-products remain.
3. Combine the remaining two partial-products by a conventional adder.

For the second step, repeat the following steps:

(a) Partial-products that feature at least three 1-bit products with the same weight, i.e. are in the same column, form a layer.

(b) In this layer, any three 1-bit products with the same weight are input into a full adder. The sum bit will carry the same weight, whereas the weight of the carry-out will increase by 1.

(c) In this layer, if two 1-bit products with the same weight remain, input them into a half adder where the weight of the sum is unaltered.

(d) If a single 1-bit product remains, pass it to the next iteration.

Figure 2.3 illustrates the steps (a-d) for a 5-bit multiplier and shows the corresponding hardware design. As it can be observed, the multiplier is composed of 11 FAs, 4 HAs, and 1 ripple carry adder (RCA). A wallace tree is a parallel addition tree, which requires $O(\log(N))$ to accumulate all partial-products [34]. The full and half adders have a delay of $O(1)$. Since the final addition, computed by the RCA, has a delay of $O(\log(N))$, the overall time complexity is only $O(\log(N))$ However, the disadvantage of a Wallace tree multiplier is the area complexity as well as the irregular structure causing difficulties in the layout, which can lead to longer wires with increased capacitance.

2.3 Multiplication and Crandall's Reduction Combined

In case of reduction with Mersenne primes, i.e. $M_p = 2^p - 1$ where p is itself a prime, we can apply the fast reduction method [30]. For Mersenne primes the following congruence relation holds:

$$2^p \equiv 1 \pmod{2^p - 1}, \tag{2.1}$$

which leads to the fast reduction procedure by writing $C = A \cdot B = C_h 2^p + C_l$ and combining it with Equation (2.1):

$$C \equiv C_h + C_l \pmod{2^p - 1}. \tag{2.2}$$

Fast reduction is commonly applied after the accumulation of the digit-products, but instead we combined both steps.

For combined reduction, all digit-product bits exceeding the Mersenne prime M_p must be shifted to the right by p bits. This can be combined with the Wallace tree construction i.e. before each layer reduction, the those exceeding bits are shifted. As illustrated in Figure 2.4, this leads to a Wallace tree design featuring a symmetric structure. Moreover, all intermediate results are of size N where $N = p - 1$; thus, a pipelined design utilizes less registers. Note that this approach enables multiplication and modular reduction being computed within the same time complexity as a Wallace tree based multiplication.

We showed that long-integer multiplication combined with Crandall's multiplication can lead to efficient multipliers that can be expressed using standard logic blocks i.e. and-gates, HAs, FAs, and RCAs. This makes our technique applicable for general hardware designs and is therefore also suitable for ASIC designs. On the contrary, FPGAs are primarily composed of LUTs, and hence a multiplier inferring standard logic gates would result in an inefficient design. Moreover, the generation of the partial-products is computed using smaller embedded multipliers, which has impact on the overall construction.

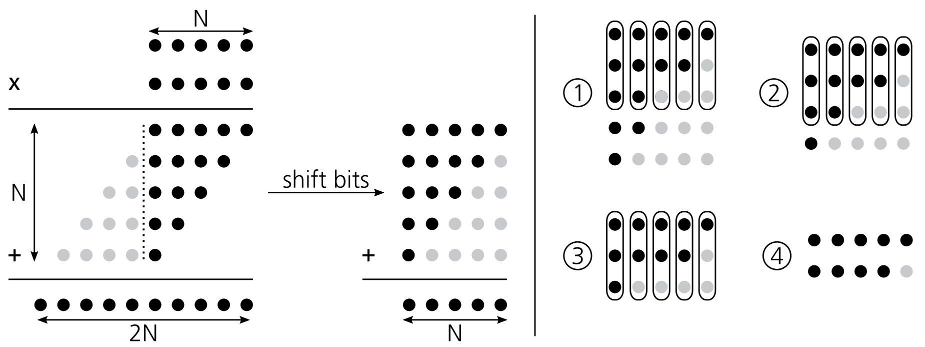


Figure 2.4: Wallace tree construction for a 5-bit multiplication combined with Crandall's reduction method.

2.4 Fast Multiplication on FPGAs

In order to realize high-performance modular multipliers on FPGAs, the following circumstances are faced: first, long integer multiplication is performed using several parallel small-sized multipliers contained in pre-fabricated DSP slices, which can operate at very high clock frequencies. Second, the modular multiplier is fully pipelined. Here, pipelining describes the process of partitioning the circuit in various stages enabling the multiplier to continuously fetch input operands while processing other multiplications. The partitioning is achieved by inserting registers at the output of all adders and multipliers. Since pipelining shortens the critical path, maximum clock frequency and throughput are increased. Third, the adder tree accumulates the digit-products of the multiplications using cascaded adders with preferably similar small sizes. The maximum clock frequency within the adder tree circuit is limited by the adder tree level containing the largest adder. As a consequence, similar sized adders lead to similar propagation delay, which enables efficient pipelining. Finally, the result is reduced using either dedicated or generic reduction techniques. While small-sized multipliers contained in DSP slices can operate at very high frequencies, the adder tree and reduction circuits are constructed with slower LUT-based FPGA logic. Hence, these latter circuits limit the performance of modular multipliers in practice. Previous works such as [1] proposed different methods to minimize the depth of the adder tree by rearranging the addition of the digit-products on digit-level. However, the resulting adder tree still suffers in performance as adders are sized differently leading to an inefficient design.

2.5 Related Work

High-performance multiplication on FPGAs is commonly performed with parallel operating small-sized multipliers that are embedded in dedicated DSP slices, each one multiplying two small-sized digits of the input operands. Asymmetric multipliers in modern FPGAs make the design of large multipliers slightly more complex as they require to decompose the input operands in asymmetric sized digits to achieve best performances. Modern synthesis tools are not always able to take this asymmetry into account and fall back on smaller symmetric multipliers. As a consequence, unnecessarily many DSP slices are instantiated which cannot be used for further functionality. Srinath and Compton [1] used asymmetric tiling techniques to exploit the capability of asymmetric multipliers and regrouped digit-products to partial-products to reduce the circuit delay of the adder tree. These techniques are summarized in the next subsections.

2.5.1 Asymmetric Tiling

Srinath and Compton [1] proposed a formalism for constructing large hardware multipliers with smaller embedded asymmetric $m{\times}n$-bit multipliers. The two input operands A and B are decomposed into smaller digits of length m and n respectively:

$$\begin{aligned} A &= [A_0, ..., A_{x-1}],\ \text{s.t. } A_i \in [0, 2^m);\ i \in [0, x)\,, \\ B &= [B_0, ..., B_{y-1}],\ \text{s.t. } B_j \in [0, 2^n);\ j \in [0, y)\,. \end{aligned}$$

Digits are then multiplied using xy $m{\times}n$-bit multiplications, where the output of each DSP multiplier is denoted by the digit-product A_iB_j:

$$A_iB_j \in [0, 2^{m+n}),\ \text{s.t. } i \in [0, x);\ j \in [0, y)\,.$$

Using symmetric multipliers typically leads to a waste of DSP resources. For instance, 64 DSP slices are required when only 17×17-bit are used to construct a 127×127-bit multiplier. On the contrary only 48 17×24 DSP slices are needed when using asymmetric tiling. Figure 2.5 depicts the multiplication of two values using asymmetric multipliers by the way of a small example, where the operand A is decomposed into $x = 2$ digits of m-bit and the operand B is decomposed into $y = 3$ digits of n-bit. The digits are multiplied together to $xy = 6$ digit-products A_iB_j. The sum over all digit-products results in the final product C. This is denoted as follows:

$$C = \sum_{j=0}^{y-1} \sum_{i=0}^{x-1} A_iB_j 2^{im+jn}\,.$$

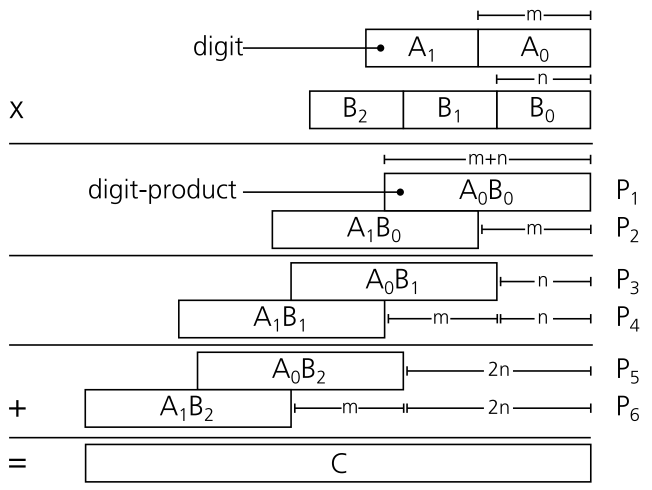


Figure 2.5: Multiplication with asymmetric tiling [1].

Digit-products A_iB_j can be seen as non-overlapping parts of partial-products P_k, i.e. each partial-product P_k is the sum of some digit-products:

$$P_k = \sum_{i,j} A_iB_j 2^{im+jn}, \text{ s.t. } \sum_k P_k = C\,.$$

An adder tree is then used to sum up all partial-products. For our example, the first stages of the adder tree may combine the partial-products P_1 and P_2, P_3 and P_4, and P_5 and P_6; and then sum up the results in succeeding adder tree levels. Accumulating the partial-products as depicted by Figure 2.5, results in an adder tree requiring 3 addition levels for 5 additions in total. The minimum adder tree level is bounded by $\lceil log_2(xy)\rceil$ [1].

2.5.2 Regrouping Digit-Products

Regrouping digit-products can reduce the depth of the adder tree and avoid unnecessary carry propagations resulting in decreased propagation delay. Figure 2.6 depicts diagonal grouping for the same configuration as in Figure 2.5. Diagonal grouping is the rearrangement of partial-products by regrouping adjacent but non-overlapping digit-products [1]. For the presented example, diagonal grouping requires only 4 partial-products, and thus the adder tree consists of only 2 adder tree levels.

Pipelining the adder tree by placing registers at the adder outputs can increase the maximum clock frequency. For an efficient design, similar delay between each adder tree level is desirable. However, adder sizes vary

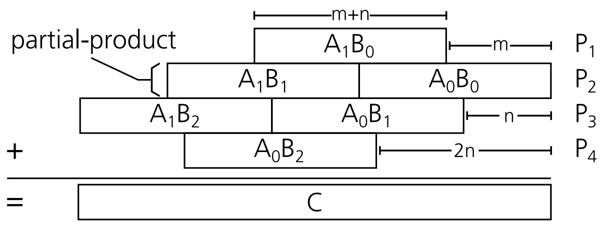


Figure 2.6: Diagonal grouping for reducing the adder tree depth [1].

for the multiplier design in Figure 2.5 as well as in Figure 2.6. For example, consider the adder tree in Figure 2.6: the two adders for summing up $P_1 + P_2$ and $P_3 + P_4$ (first adder tree level) are clearly much smaller than the adder for summing up $(P_1 + P_2) + (P_3 + P_4)$ (second adder tree level). As a consequence, the maximum frequency is limited by the relatively long propagation delay of the second level. In our modular multiplier design, we demonstrate how adder sizes can be equalized by combining the reduction with the accumulation of the digit-products.

2.6 Design Automation for Combined Reduction

We begin by formalizing the generation of the adder tree, divided in digit-product generation, digit-product splitting and partial-product generation. We describe algorithms that can be implemented by a script to automatically generate modular multipliers for variable Mersenne primes without being limited to specific DSP properties, i.e. for any $m \times n$-bit multipliers. Our resulting adder tree features equalized adder sizes which ease pipelining, and hence allows higher clock frequencies for increased performance. Finally, we embed our adder tree in a multiplier architecture that is optimized towards high-throughput and low-latency.

2.6.1 Digit-Product Generation

To begin, the position of the digit-products within the adder tree must be determined. Before combining fast reduction with digit-product accumulation, we use asymmetric tiling to compile a set of 4-tuples (i, j, μ_l, μ_h). Here i and j identify the indices of the input digits A_i and B_j, and hence connect the 4-tuple to the respective embedded multiplier in the DSP slice. The elements μ_l and μ_h denote the lower and higher bit position within the adder tree. Algorithm 1 (gen_dp) describes the digit-product generation. We

Algorithm 1 gen_dp: Determine the position of the digit-products.

```
Input: $A = [A_0, ..., A_{x-1}]$, s.t. $A_i \in [0, 2^m)$; $i \in [0, x)$ $B = [B_0, ..., B_{y-1}]$, s.t.
    $B_j \in [0, 2^n)$; $j \in [0, y)$
Output: $\mathcal{T} = \{(i, j, \mu_l, \mu_h)\}$
1: for $j$ from 0 to $y - 1$ do
2:     for $i$ from 0 to $x - 1$ do
3:         $\mu_l \leftarrow im + jn$                              ▷ lowest bit
4:         $\mu_h \leftarrow (i + 1)m + (j + 1)n - 1$              ▷ highest bit
5:         $\mathcal{T} \leftarrow \mathcal{T} \cup \{(i, j, \mu_l, \mu_h)\}$    ▷ add tuple
6:     end for
7: end for
8: return $\mathcal{T}$
```

assume that the input operands A and B are decomposed by m and n respectively. The output of Algorithm 1 (gen_dp) is a set $\mathcal{T}$ storing instances of the 4-tuple. An exemplary 4-tuple is depicted in the upper left corner of Figure 2.7.

2.6.2 Digit-Product Splitting

For combined reduction, all digit-product bits exceeding the Mersenne prime M_p must be shifted to the right by p bits. Digit-product bits exceeding the position $2p$ are unused and set to 0. As a consequence, they do not contribute to the multiplier result and can be removed (marked by dark-grey boxes in Figure 2.7). In addition, digit-products should be regrouped to reduce the adder tree depth (see Section 2.2). The steps described above are inefficient when performed on the digit-product data structure. For example, digit-products that partly exceed the Mersenne prime M_p need to be split in two parts. The upper part of the digit-product is then shifted to the right by p bits whereas the lower part remains unaltered. Instead, we suggest to perform the shifting for fast reduction and the subsequent regrouping on bit-level. Therefore, it is required to disassemble all digit-products i.e. instances of 4-tuples contained in $\mathcal{T}$ bit-wise. Algorithm 2 (slice_dp) performs this procedure and also shifts the corresponding bits for combined reduction and removes unused ones. Bits are described by another 4-tuple described by (i, j, μ_a, μ_r). The identifiers i and j are inherited from the respective digit-product. The absolute bit position μ_a represents the position within the adder tree, whereas μ_r describes the relative bit position within a digit-product. Storing the relative bit position μ_r is required for implementation purposes, because it

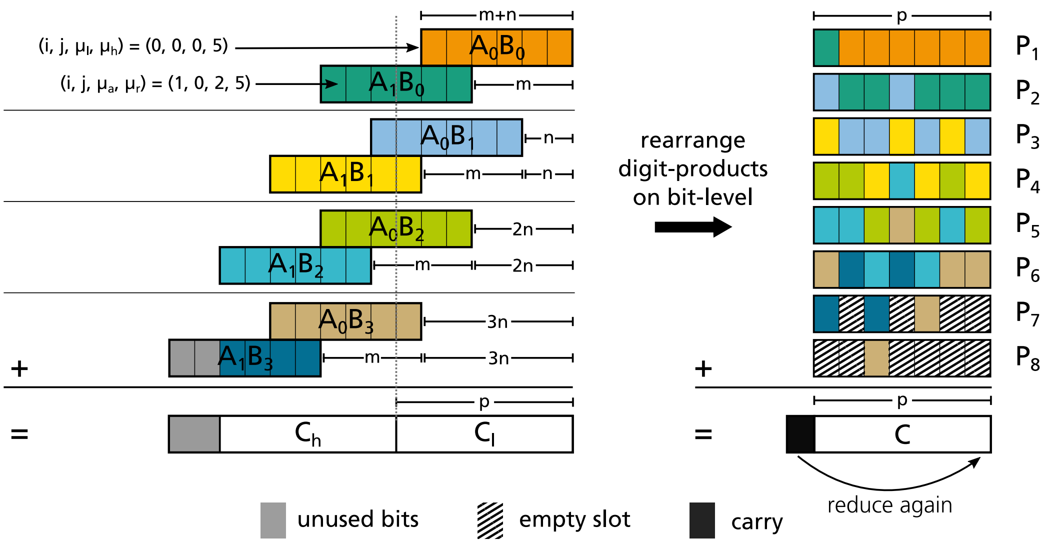


Figure 2.7: Adder tree optimized towards high-performance. Left: Digit-products generated for $m = 4$, $n = 2$ and $M_7 = 2^7 - 1$. Right: Rearranged sliced digit-products to partial-products with combined fast reduction.

Algorithm 2 slice_dp: Slice digit-products in single bits.

Input: $\mathcal{T} = \{(i, j, \mu_l, \mu_h)\}$, $M_p = 2^p - 1$
Output: $\mathcal{Z} = \{(i, j, \mu_r, \mu_a)\}$

```
 1: for each $t$ in $\mathcal{T}$ do
 2:     $(i, j, \mu_l, \mu_h) \leftarrow t$
 3:     for $k$ in 0 to $(\mu_h - \mu_l)$ do
 4:         $v \leftarrow \mu_l + k$
 5:         if $v < 2p$ then
 6:             $\mu_r \leftarrow k$                                  ▷ relative
 7:             $\mu_a \leftarrow v \bmod p$                          ▷ absolute
 8:             $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{(i, j, \mu_r, \mu_a)\}$   ▷ add tuple
 9:         end if
10:     end for
11: end for
12: return $\mathcal{Z}$
```

enables to associate each bit with the correct DSP multiplier output. The output of Algorithm 2 is a set $\mathcal{Z}$ storing instances of bits represented by the respective 4-tuple. Figure 2.7 illustrates digit-product slicing together with shifted and removed bits.

2.6.3 Rearrange Sliced Digit-Products

In the last step, all bits are assigned to partial-products. Each partial-product P_k is represented by a set $\mathcal{P}_k$ holding instances of the 4-tuple (i, j, μ_a, μ_r). Instances of this 4-tuple are assigned to partial-products as follows: We create a new partial-product and iterate from the absolute bit position 0 to $p-1$ such that a partial-product contains at most p instances of the 4-tuple. Whenever an unassigned 4-tuple with correct absolute bit position is found, it is added to the corresponding partial-product. Once we iterated through all bit positions and unassigned 4-tuples still remain, a new partial-product is created. The corresponding procedure is illustrated in Algorithm 3 (rearrange_dp). The number of created partial products is given by

Algorithm 3 rearrange_dp: Rearrange sliced digit-products.

```
Input: $\mathcal{Z} = \{(i, j, \mu_r, \mu_a)\}$, $M_p = 2^p - 1$
Output: $\{\mathcal{P}_k\}_{k \geq 1}$
 1: $k \leftarrow 0$
 2: while $\mathcal{Z} \neq \emptyset$ do
 3:     $k \leftarrow k + 1$
 4:     for $v$ in 0 to $p - 1$ do                          ▷ iterate bit positions
 5:         if $z$ in $\mathcal{Z}$ with $\mu_a = v$ then           ▷ select tuple
 6:             $\mathcal{P}_k \leftarrow \mathcal{P}_k \cup \{(i, j, \mu_r, \mu_a)\}$            ▷ add tuple
 7:             $\mathcal{Z} \leftarrow \mathcal{Z} \setminus \{(i, j, \mu_r, \mu_a)\}$             ▷ remove tuple
 8:         end if
 9:     end for
10: end while
11: return $\{\mathcal{P}_k\}$
```

the maximum number of tuples that feature the same absolute bit position. Since our algorithm assigns a tuple whenever possible, it can be guaranteed that the lowest number of possible partial products is obtained. With information contained in partial-products $\mathcal{P}_k$, one can construct the hardware description of the respective modular multiplier. The right part of Figure 2.7 depicts the rearrangement of sliced digit-products to partial-products. It can be observed that our approach equalizes the size of the corresponding partial-products. The former adder tree features adder sizes up to $2p$, whereas our

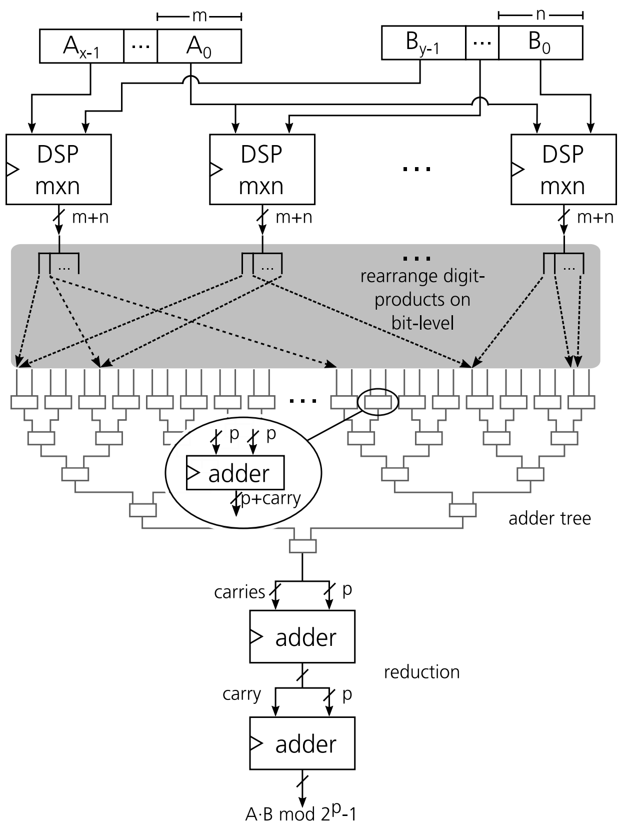


Figure 2.8: Hardware architecture of high-performance modular multiplier using optimized adder tree.

optimized adder tree features a maximum adder size of p plus some carry bits depending on the number of adder tree levels. Reduced and equalized adder sizes allow a higher maximum clock frequency which translates to increased throughput and reduced latency. Note that two further additions are required after accumulating all partial-products due possible carry bits.

2.7 Hardware Design and Analysis

Figure 2.8 depicts the hardware architecture of our modular multiplier for Mersenne primes. The hardware architecture is divided in four parts: the multiplication of digits using DSP multipliers, the subsequent rearranging of sliced digit-products to partial-products, the summation of partial-products, and finally the two addition steps for full reduction. For high-performance purposes, DSP slices compute digit-products fully parallel. Furthermore, we

Table 2.1: Comparison with related work of area utilization and performance for modular multiplication in $\mathbb{F}_p$ with $p = 2^{127} - 1$. Note that [1] excludes the reduction.

Work	**CC**	**Freq. (MHz)**	**TP (GBit/s)**	**Latency (ns)**	**Resources**			
					DSP	**Slices**	**LUT**	**Reg.**
[36]	31	110	0.45	281.82	4	1139	-	-
[35]	20	190	3.45	105.26	16	-	-	-
[1]	5	115	14.55	43.64	48	513	1703	2076
This work	**7**	**156**	**19.81**	**44.87**	**48**	**547**	**1821**	**2169**

make use of the registers that are embedded in each DSP slice. After all digit-products are obtained, the sliced DSP multiplier outputs are rearranged as discussed in previous sections. All single bits are grouped to partial-products which are then summed up with the subsequent adder tree. The rearrangement of digit-products to partial-products has no impact on area because it only translates to signal rewiring. We can pipeline the adder tree efficiently because all adders are similar sized translating to an equivalent circuit delay between register stages. With each adder tree level, the input size is increased by 1-bit corresponding to the carry of the previous addition. Once the accumulation of all partial-products is completed, two extra additions are performed for full reduction. Finally, the result matches the modular multiplication i.e. $A \cdot B \bmod 2^p - 1$.

2.7.1 Results

We developed a script that performs the regrouping of digit-products for arbitrary Mersenne primes and DSP multiplier widths. Our script also generates the multiplier's hardware description in VHDL including test vectors and test benches. We have implemented, synthesized and simulated our VHDL code with Xilinx Vivado 2016.2. All our synthesis results were obtained after place-and-route using default synthesis and implementation strategies. Table 2.1 illustrates a comparison of area utilization and performance in the case of $M_{127} = 2^{127} - 1$ with related work [1, 35, 36]. We implemented our design on a Xilinx's Zynq-7020 FPGA because it is widely used in the research community. It is also used by state of the art ECC implementations (e.g. Järvinen et al. [35] or Sasdrich and Güneysu [37]). Srinath and Compton [1] used a Virtex-5 and did not include the reduction procedure. For a fair comparison, we implemented their work on the Zynq-7020 platform and pipelined

Table 2.2: Area utilization and performance results of our proposed multiplier for various Mersenne primes.

FPGA	**Mers. Prime**	**Multiplier Width**	**CC**	**TP (GBit/s)**	**Latency (ns)**	**Resources DSP**	**Resources Slices**
Zynq-7020	$2^{61}-1$	72×68	7	15.25	28.00	12	158
Zynq-7020	$2^{89}-1$	96×102	7	17.70	35.20	24	333
Zynq-7020	$2^{107}-1$	120×119	7	18.38	40.75	35	439
Zynq-7020	$2^{127}-1$	144×136	7	19.81	44.87	48	547
Zynq-7045	$2^{61}-1$	72×68	7	27.73	15.40	12	157
Zynq-7045	$2^{89}-1$	96×102	7	33.59	18.55	24	306
Zynq-7045	$2^{107}-1$	120×119	7	35.67	21.00	35	428
Zynq-7045	$2^{127}-1$	144×136	7	39.69	22.40	48	546
Zynq-7045	$2^{521}-1$	528×527	9	55.42	84.60	682	7527

their design. Compared to [1], our modular multiplier achieves a 1.3-factor improvement in throughput (TP), while featuring very low-latency. This improvement is linked to the adder size reduction and equalization, which results in an adder tree where each adder tree level operates at its maximum clock frequency. We also note that the cycle count (CC) of our implementation has improved, however, our DSP utilization is 4-times and 12-times higher than when compared to [35] and [36], respectively.

While M_{127} is applied in today's cryptography, other Mersenne primes might receive more attention in the future. Therefore, we further report implementation results for Mersenne primes between M_{61} and M_{127} in Table 2.2. We also implemented our design on the high-end FPGA Zynq-7045, where we synthesized our modular multiplier for the Mersenne prime M_{521} [38].

2.8 Conclusions

In this chapter, we presented a novel hardware design for the multiplication in Mersenne prime fields based on a new optimization strategy of the adder tree and reduction circuits at the bit-level. On an FPGA, the presented modular multiplier can operate at higher frequencies, leading to improved throughput and latency. We provided a formalization of our proposed strategy for any Mersenne prime and any size of the underlying small-sized (potentially asymmetric) multipliers.

Chapter 3

Elliptic Curve Cryptography

In this chapter, the fundamentals of elliptic curve cryptography (ECC) are explained, which shall aid the reader in understanding the implementation of the Montgomery curve Curve25519 [12] in the next chapter. We describe the elliptic curve scalar multiplication, which is the core of all elliptic curve cryptosystems, and construct the elliptic curve Diffie-Hellman (ECDH) key exchange. Moreover, we derive the required formulas for point addition and point doubling in affine and projective coordinates. For a more detailed description we refer the reader to [39, 40].

Outline. Section 3.1 provides the definition of elliptic curves. This is followed by a description of the scalar multiplication in Section 3.2. Section 3.3 gives details on affine and projective coordinates and Section 3.4 presents the Montgomery ladder as an efficient time-constant algorithm for scalar multiplication.

3.1 Introduction to Elliptic Curves

An elliptic curve E over a field K is defined by the *Weierstrass* equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{3.1}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and $\Delta \neq 0$, where Δ is the discriminant of E. The discriminant Δ of E is the quantity

$$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \in K$$

where

$$\begin{aligned}
b_2 &= a_1^2 + 4a_2 \\
b_4 &= a_1 a_3 + 2a_4 \\
b_6 &= a_3^2 + 4a_6 \\
b_8 &= a_1^2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 + 4a_2 a_6 - a_4^2 \,.
\end{aligned}$$

The condition $\Delta \neq 0$ ensures that no points exist that have more than one tangent i.e. a curve is said to be smooth. Sometimes E/K is written to emphasize that E is defined over K and K is the underlying field. If L is any extension field of K, then the set of L-rational points on E is

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1 xy + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6\} \cup \{\infty\}$$

where ∞ is the *point-at-infinity*, which can be seen as the point that is intersected by all lines parallel to the y-axis. The L-rational points on E are the points (x, y) that satisfy the curve equation and whose coordinates are in L.

3.1.1 Short Weierstrass Form

If the characteristic of the field K is $\mathrm{char}(K) \neq (2, 3)$ then the general Weierstrass Equation (3.1) can be simplified to the *short Weierstrass* form [41]

$$E : y^2 = x^3 + ax + b \,, \tag{3.2}$$

where $a, b \in K$ and $\Delta = 4a^3 + 27b^2 \neq 0$. In this thesis, we focus on curves defined over prime fields $\mathbb{F}_p$ with $p > 3$ and thus $\mathrm{char}(\mathbb{F}_p) > 3$. Then the set of points that satisfy Equation (3.2) including the point-at-infinity is given by

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : y^2 - x^3 - ax - b = 0\} \cup \{\infty\} \,.$$

3.2 Scalar Multiplication and the Elliptic Curve Diffie-Hellman Key Exchange

The set $E(\mathbb{F}_p)$ together with the *chord-and-tangent rule* (see the next section for further details), which represents the group operation, forms an abelian additive group $(E(\mathbb{F}_p), \oplus)$. Note that the point-at-infinity ∞ acts as the identity or neutral element $\mathcal{O}$. The group can be used to construct an elliptic curve cryptosystem. Let $P \in E(\mathbb{F}_p)$ be a point of order r, then the cyclic subgroup of $E(\mathbb{F}_p)$ generated by P is $\{\mathcal{O}, P, 2P, ...(r-1)P\}$. Then the order of a point P corresponds to the cardinality of the generated cyclic subgroup. Moreover, if the order of the group $E(\mathbb{F}_p)$ is prime, then every point except the identity element is a generator of this group. This can be deduced by Lagrange's theorem, which states that the order of a subgroup H of group G divides the order of G.

With an integer $k \in [1, r-1]$, the *point multiplication* or *scalar multiplication* describes the operation of adding a point P to itself $(k-1)$-times:

$$Q = [k]P = \underbrace{P \oplus P \oplus ... \oplus P}_{k-1 \text{ additions}},$$

where the result Q is also a point in the subgroup of $(E(\mathbb{F}_p), \oplus)$ generated by P.

The scalar multiplication serves as the trapdoor function and is comparable to the exponentiation operation in Section 1.1. Analog to the exponentiation operation of a conventional DH key exchange, the scalar multiplication enables to construct ECDH for prime fields as follows:

1. Alice and Bob agree on (E, p, P) where E is the elliptic curve, p describes the prime field $\mathbb{F}_p$, and P is the base point.
2. Alice chooses $k_A \leftarrow \mathbb{Z}_q$ uniformly at random, and computes $h_1 := [k_A]P$. Then she sends h_1 to Bob.
3. Bob receives h_1. He chooses $k_B \leftarrow \mathbb{Z}_q$ uniformly at random and computes $h_2 := [k_B]P$. Bob sends h_2 to Alice and outputs the key $k_{AB} := [k_B]h_1 = [k_B][k_A]P$.
4. Alice receives h_2 and outputs the key $k_{AB} := [k_A]h_2 = [k_A][k_B]P$.

3.2.1 Group Law on Elliptic Curves

To obtain a group structure, it is required to define a group operation for $E(\mathbb{F}_p)$. This group operation is geometrically described by the *chord-and-tangent* rule. Let two points be denoted by $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ where $P, Q \in E(\mathbb{F}_p)$. The *point addition* operation is denoted by $P \oplus Q$ and geometrically obtained by projecting the point over the x-axis that is intersected by the line that connects P and Q. For the addition of a point $P = (x, y)$ and its opposite $\overline{P} = (x, -y)$, which is the projection of P over the x-axis, the corresponding line intersects the curve in the *point-at-infinity* $P \oplus \overline{P} = \infty$. The opposite point $\overline{P}$ can be also denoted by $-P$. Adding a point to itself, i.e. $P \oplus P$ is known as the *point doubling* operation, where the line becomes a tangent to P which intersects E in a second point. The point

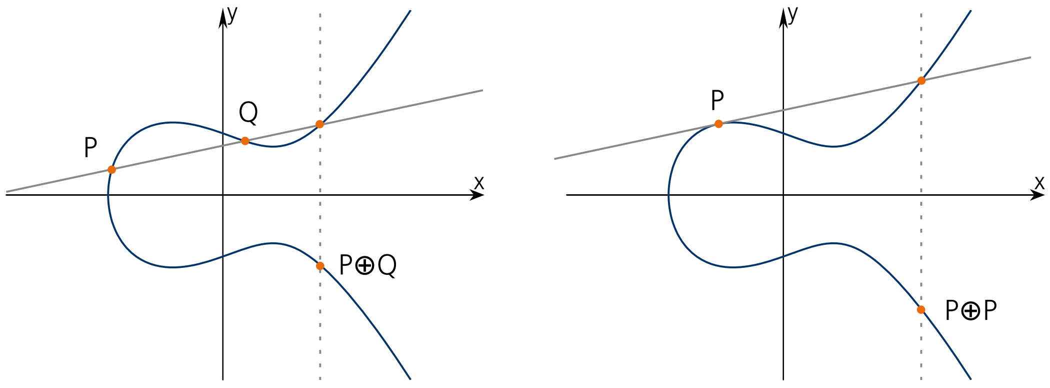


Figure 3.1: ECC group law on $E: y^2 = x^3 - x + 1$ over the field $\mathbb{R}$. Left: Point addition. Right: Point doubling.

doubling operation $P \oplus P$ is often written as $2P$. Figure 3.1 illustrates the point addition and point doubling operation for an exemplary elliptic curve $E: y^2 = x^3 - x + 1$ over $\mathbb{R}$.

3.3 Choice of Coordinates

To derive a mathematical description of the point addition and point doubling operation, the described steps of the chord-and-tangent rule are made explicit for the corresponding coordinates. We begin with the intuitive affine coordinates, followed by projective coordinates for improved performance.

3.3.1 Affine Coordinates

Let $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$ such that $P \neq \pm Q$ and $P \oplus Q = (x_{P\oplus Q}, y_{P\oplus Q})$. In this case, the point addition is given by:

$$x_{P\oplus Q} = \lambda^2 - x_P - x_Q\,, \quad y_{P\oplus Q} = \lambda(x_P - x_{P\oplus Q}) - y_P\,, \quad \lambda = \frac{y_P - y_Q}{x_P - x_Q}\,.$$

Let $2P = (x_{2P}, y_{2P})$, then point doubling is given by:

$$x_{2P} = \lambda^2 - 2x_P\,, \quad y_{2P} = \lambda(x_P - x_{2P}) - y_P\,, \quad \lambda = \frac{3x_P^2 + a}{2y_P}\,.$$

We note that for point addition and point doubling $I + 2M + S$ and $I + 2M + 2S$ operations are required, respectively, where I stands for modular inversion, M for modular multiplication, and S for modular squaring.

Note that the modular inversion operation is relatively complex as it is composed of numerous modular multiplications and squarings (e.g. an inversion in $\mathbb{F}_p$ where $p = 2^{255} - 19$ needs $254S + 11M$ based on Fermat's little theorem). Instead, points on a curve can be represented in projective coordinates, which avoid the costly inversion for point addition and doubling.

3.3.2 Projective Coordinates

In projective coordinates, a point is represented by $(X : Y : Z)$ on E following the relation $x = X/Z,\ y = Y/Z$ with $Z \neq 0$. The set of all projective points is denoted by $\mathbb{P}(\mathbb{F}_p)$. Projective coordinates are unique up to multiplication by non-zero elements, which is denoted by the equivalence relation $(X : Y : Z) = \{(\lambda X, \lambda Y, \lambda Z) : \lambda \in \mathbb{F}_p\}$. The set of projective points

$$\mathbb{P}(\mathbb{F}_p)^0 = \{(X : Y : Z) : X, Y, Z \in \mathbb{F}_p,\ Z = 0\}$$

is called the *line at infinity*. Though, the only point on the line at infinity that also lies on E is $(0 : \lambda : 0)$, which corresponds to ∞.

The point opposite to $(X : Y : Z)$ is denoted by $(X : -Y : Z)$. Based on those notations, the elliptic curve *short Weierstrass* Equation (3.2) is changed to

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \,.$$

Let $P = (X_P : Y_P : Z_P),\ Q = (X_Q : Y_Q : Z_Q)$ such that $P \neq \pm Q$ and $P \oplus Q = (X_{P\oplus Q} : Y_{P\oplus Q} : Z_{P\oplus Q})$. Then set

$$A = Y_QZ_P - Y_PZ_Q \,, \quad B = X_QZ_P - X_PZ_Q \,, \quad C = A^2Z_PZ_Q - B^3 - 2B^2X_PZ_Q \,,$$

and the *point addition* operation $P \oplus Q$ is described by:

$$X_{P\oplus Q} = BC \,, \quad Y_{P\oplus Q} = A(B^2X_PZ_Q - C) - B^3Y_PZ_Q \,, \quad Z_{P\oplus Q} = B^3Z_PZ_Q \,.$$

Let $2P = (X_{2P} : Y_{2P} : Z_{2P})$, then *point doubling* is given by:

$$A = aZ_P^2 + 3X_P^2 \,, \quad B = Y_PZ_P \,, \quad C = X_PY_PB \,, \quad D = A^2 - 8C \,,$$

and

$$X_{2P} = 2BD \,, \quad Y_{2P} = A(4C - D) - 8Y_P^2B^2 \,, \quad Z_{2P} = 8B^3 \,.$$

Compared to affine coordinates, the modular inversion is omitted and the computation requires $12M + 2S$ operations for point addition and $7M + 5S$ operations for point doubling.

Fast Montgomery Arithmetic

Montgomery [4] further improved the addition and doubling formulas for a special type of curve, i.e. the *Montgomery form*, that is

$$E_M : By^2 = x^3 + Ax^2 + x \, . \tag{3.3}$$

The Montgomery arithmetic relies on an efficient x-coordinate only computation based on the x-coordinate of the two points $x(P), x(Q)$ and the x-coordinate of the difference point $x(P \ominus Q) = x(P \oplus \overline{Q}) = x(P \oplus -Q)$, all in projective coordinates. In Section 3.4, it is shown how this representation can be used to describe an efficient scalar multiplication algorithm. For differential-addition we write:

$$\begin{aligned} X_{P\oplus Q} &= Z_{P\ominus Q}((X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q))^2 \, , \\ Z_{P\oplus Q} &= X_{P\ominus Q}((X_P - Z_P)(X_Q + Z_Q) - (X_P + Z_P)(X_Q - Z_Q))^2 \, . \end{aligned}$$

For doubling we write:

$$\begin{aligned} 4X_P Z_P &= (X_P + Z_P)^2 - (X_P - Z_P)^2 \, , \\ X_{2P} &= (X_P + Z_P)^2 (X_P - Z_P)^2 \, , \\ Z_{2P} &= 4X_P Z_P((X_P - Z_P)^2 + ((A+2)/4)(4X_P Z_P)) \, . \end{aligned}$$

Hence, an x-coordinate point addition and doubling is computed in $4M + 2S$ and $3M + 2S$ operations.

3.4 Montgomery Ladder

A scalar multiplication $[k]P$ can be easily computed by adding the point P to itself k-times, however, the computational complexity would grow exponential with increasing size of k, where the size of k in number bits is given by $|k|$. Instead, a point is multiplied with a scalar by combining point addition and point doubling operations. Algorithm 4 (mont_ladder) depicts the generalized Montgomery ladder [4] that can be applied for any group (even though we use a curve group $(E, \oplus)$ in our example). It ensures that the same arithmetic operations are executed independently of the scalar bit k_i and is therefore a common algorithm used in designs of constant-time implementations. It is further assumed that the most significant bit of k is set to 1, i.e. $k \in [2^{|k|-1}, 2^{|k|})$ From Line 4 and 6 it can be observed that the difference point $R_2 \ominus R_1$ remains constant. From Line 1 we observe that the difference is P, and hence:

$$R_2 = R_1 \oplus P \, . \tag{3.4}$$

Algorithm 4 mont_ladder: The classical Montgomery ladder [4].

```
Input: ($k = \sum_{i=0}^{|k|-1} k_i 2^i$) $\in (2^{|k|-1}, 2^{|k|}]$, $P \in E$.
Output: $Q \leftarrow [k]P$.
 1: $R_1 \leftarrow \mathcal{O}$ and $R_2 \leftarrow P$
 2: for $i = |k| - 1$ downto 0 do
 3:     if $k_i = 0$ then
 4:         $R_1 \leftarrow 2R_1$ and $R_2 \leftarrow R_1 \oplus R_2$
 5:     else
 6:         $R_1 \leftarrow R_1 \oplus R_2$ and $R_2 \leftarrow 2R_2$
 7:     end if
 8: end for
 9: $Q \leftarrow R_1$
10: return $Q$
```

Moreover, the following relation can be determined:

$$R_1 = [(k)_i]P, \quad R_2 = [(k)_i + 1]P, \quad \text{where } (k)_i := \lfloor k/2^i \rfloor .$$

Based on Equation (3.4), the correctness of Algorithm 4 can be observed by relating it to the standard double-and-add algorithm. If $k_i = 0$, then R_1 is doubled. If $k_i = 1$, then R_1 is replaced by $R_1 \oplus R_2$. Combined with Equation (3.4), we get $R_1 \oplus R_2 = R_1 \oplus R_1 \oplus P = 2R_1 \oplus P$, which means we double R_1 and add P. Since the difference point is known and remains constant, the differential addition formulas in projective coordinates from the previous section can be embedded into the Montgomery ladder.

Chapter 4

X25519 DH Key Exchange on an FPGA

In this chapter, we present a low-latency X25519 hardware implementation, which is the DH key exchange based on Curve25519 [12, 42]. This is achieved by using an extended version of the high-speed modular multiplier we presented in Chapter 2. Our implementation uses the Montgomery ladder as the scalar multiplication algorithm and includes randomized projective coordinates to thwart DPA attacks. Parts of this chapter have been published in [26, 27].

Outline. Section 4.1 provides background information on Curve25519. Section 4.2 presents an algorithmic description of the X25519 key exchange. Section 4.3 summarizes the implemented field arithmetic and Section 4.4 the corresponding hardware design. Section 4.5 presents the synthesis and performance results. Finally, we conclude in Section 4.6.

4.1 Background

Curve25519 is a 128-bit secure elliptic curve introduced by Bernstein in 2006 [12]. It is designed in an elegant and transparent way, while offering high-performance, which makes it a promising candidate to secure IoT applications. Over the past few years, numerous high-speed Curve25519 implementations on embedded devices have been published in literature [15, 43, 44]. The fastest implementation on microcontrollers to date performs a variable-base scalar multiplication on an ARM Cortex-M4 microcontrollers in 1423667 cycles [45]. An *application specific instruction set processor* for IoT applications has been presented in [46], where a session key can be computed in between 811170 and 3455394 clock cycles, depending on area and power constraints. Nowadays, embedded devices often possess additional reconfigurable hardware logic, that can be used for accelerating cryptographic operations. Examples of these embedded devices are Xilinx's Zynq-7000 [47], Altera's Aria V, Cyclone V and Stratix 10 [48], and Microsemi's SmartFusion and SmartFusion2. Sasdrich and Güneysu [49] were the first to present a hardware based Curve25519 implementation optimized for high-throughput applications on Xilinx Zynq-7020 devices. Sasdrich and Güneysu's implementation requires 34052 cycles at a maximum frequency of 100 MHz for one Curve25519 scalar multiplication and thus, a session key is computed in about 340 ms. Their design is based on multiple parallel cores, each one performing one elliptic curve scalar multiplication in projective coordinates, and achieves a throughput of 32000 scalar multiplications per second on a Xilinx Zynq-7020 at 100 MHz. Sasdrich and Güneysu suggested randomized projective coordinates as a side-channel countermeasure to thwart differential power analysis in the extended version of their paper [37].

Our implementation uses the Montgomery ladder in projective coordinates [4] to perform a variable-base scalar multiplication using Curve25519 to realize the ECDH key exchange protocol. To reduce the latency of a scalar multiplication, we use a high-speed optimized prime field multiplier. To thwart differential side-channel attacks we show that our design can inherently make use of randomized projective coordinates at no extra area costs and with only a negligible time overhead [50, 51]. Our implementation performs one Curve25519 scalar multiplication in 10,465 cycles at a frequency of 115 MHz on a Xilinx Zynq-7030 and 84 MHz on a Zynq-7020, hence a session key is computed in 92 μs and 125 μs, respectively. The former constitutes an improvement of 1.3 compared to our work in [26].

4.2 Algorithmic Description

The ECDH Curve25519 key exchange protocol (also known as X25519 [52]) allows two parties to derive a shared session key using Curve25519. Curve25519 is a Montgomery elliptic curve [4] (compare Equation (3.3) from Chapter 3) defined by the equation:

$$E_M \;:\; y^2 = x^3 + 486662x^2 + x\,, \tag{4.1}$$

over the prime field $\mathbb{F}_{2^{255}-19}$. The set of points $\{(x, y) \in \mathbb{F}^2_{2^{255}-19} : y^2 = x^3 + 486662x^2 + x\}$ together with the point-at-infinity ∞ serving as neutral element $\mathcal{O}$ forms an additive abelian group under point addition. In order to compute a shared key between two parties, a public point P on E is added to itself $|k| - 1)$ times, where k is a 255-bit secret private value. According to [12] the three least significant bits of the scalar are set to 0 to overcome small-subgroup attacks.

As described in Algorithm 4 (mont_ladder) from Section 3.4, an efficient way to compute a scalar multiplication on Montgomery elliptic curves is given by the Montgomery powering ladder. The explicit Montgomery ladder algorithm for Curve25519 using randomized projective coordinates is shown in Algorithm 5 (x25519_ladder). Thereby, we use the x-only coordinates from Section 3.3.2 together with differential addition $x(P \oplus Q)$ and doubling formulas $x(2P)$. According to [50], randomized projective coordinates are a countermeasure to thwart side-channel attacks by randomly projecting the input point P, i.e. $(X, Y, Z) = (\lambda X, \lambda Y, \lambda Z)$ for a random value $\lambda \in \mathbb{Z}_2^{255} \backslash \{0\}$. Moreover, x25519_ladder replaces the conditional branch in mont_ladder by a *conditional-swap* function to prevent timing attacks. In every ladder iteration a conditional swap of the points R_1, R_2 is performed depending on the value of the secret bit k_i followed by a point addition and a point doubling operation. Finally, the result of the ladder is transformed back to the original domain. This operation requires a modular inversion (Line 27) and one extra multiplication.

4.3 Pseudo Mersenne Prime Field Arithmetic

In the following, the implementation of the arithmetic modules, i.e. modular addition, modular subtraction, and modular multiplication, is analyzed. These modules are frequently accessed by the Montgomery ladder and thus, they contribute strongly to the overall performance. Then the design of the Montgomery ladder is presented, where about 80 % of the time is spent for scalar multiplication [49]. Afterwards, we describe the logic for the modular inversion based on Fermat's little theorem.

4.3.1 Addition and Subtraction

The Montgomery ladder requires computations in the field $\mathbb{F}_p$, i.e. the implementation of integer arithmetic modulo p. Let x and y be two n-digit radix-b positive integers $0 \leq x, y < p$, then modular addition can be implemented trivially by adding digit by digit while rippling the carry bit over the partial sums. The reduction operation is performed subsequently to the addition

Algorithm 5 x25519_ladder: Curve25519 Montgomery ladder in randomized projective coordinates.

```
Input: $\left(k = \sum_{i=0}^{254} k_i 2^i\right) \in [2^{254}, 2^{255})$, $\lambda \in \mathbb{Z}_2^{255} \backslash \{0\}$, and $x(P) \in E$.
Output: $x(Q) \leftarrow x([k]P)$
 1: $R_1 = (X_1, Z_1) \leftarrow (\lambda, 0)$                     ▷ cf. Algorithm 4, $R_1 \leftarrow \mathcal{O}$
 2: $R_2 = (X_2, Z_2) \leftarrow (\lambda \cdot x(P), \lambda)$              ▷ cf. Algorithm 4, $R_2 \leftarrow P$
 3: $R_2 \ominus R_1 = (X_3, Z_3) \leftarrow (\lambda \cdot x(P), \lambda)$
 4: for $i = 254$ downto 0 do
 5:     $(R_1, R_2) \leftarrow$ cswap$(k_i \oplus k_{i+1}, (R_1, R_2))$
 6:     $t_1 \leftarrow X_1 + Z_1$
 7:     $t_2 \leftarrow X_1 - Z_1$
 8:     $t_3 \leftarrow X_2 + Z_2$
 9:     $t_4 \leftarrow X_2 - Z_2$
10:     $t_6 \leftarrow t_1^2$
11:     $t_7 \leftarrow t_2^2$
12:     $t_5 \leftarrow t_6 - t_7$
13:     $t_8 \leftarrow t_4 t_1$
14:     $t_9 \leftarrow t_3 t_2$
15:     $t_{10} \leftarrow t_8 + t_9$
16:     $t_{11} \leftarrow t_8 - t_9$
17:     $t_{15} \leftarrow t_{10}^2$
18:     $X_2 \leftarrow Z_3 t_{15}$
19:     $t_{12} \leftarrow t_{11}^2$
20:     $t_{13} \leftarrow 121666 t_5$
21:     $X_1 \leftarrow t_6 t_7$
22:     $t_{14} \leftarrow t_7 + t_{13}$
23:     $Z_2 \leftarrow X_3 t_{12}$
24:     $Z_1 \leftarrow t_5 t_{14}$
25: end for
26: $(R_1, R_2) \leftarrow$ cswap$(k_0, (R_1, R_2))$
27: $Z_1 \leftarrow Z_1^{-1}$                          ▷ transform back to affine coordinates
28: $x(Q) \leftarrow X_1 Z_1$
29: return $x(Q)$
```

by applying Crandall's fast reduction [30] i.e. adding the potential carry bit on the earlier obtained addition/subtraction result. Modular subtraction follows a similar procedure. More details can be found in [53]. Centerpiece of the modular addition and modular subtraction unit, computing $x \pm y \bmod p$, are the 255-bit wide addition, respectively 255-bit wide subtraction blocks. Both, addition and subtraction, can be executed in 1 clock cycle. For the reduction procedure 1 cycle is needed additionally and hence, a total of 2 cycles is required for modular addition/subtraction.

4.3.2 Multiplication

As proposed in Chapter 2, high-speed modular multipliers can be efficiently implemented with parallel operating DSP slices and an optimized adder tree that interleaves the fast reduction procedure. We extended this approach and implemented a similar modular multiplier for the pseudo Mersenne prime $2^{255} - 19$. In case of reduction with a pseudo Mersenne primes, i.e. $p = 2^n - c$ where n is a positive integer and c is a constant, the following congruence relation holds:

$$2^n \equiv c \pmod{2^n - c},$$

which leads to the fast reduction procedure by writing $C = A \cdot B = C_h 2^n + C_l$ and applying the previous congruence relation:

$$C \equiv C_h c + C_l \pmod{2^n - c}.$$

An additional multiplication by the constant c is required before the reduction can take place. This multiplication by a constant increases the area utilization, but does not result in an extra clock cycle because it is combined with the digit-product generation. Overall, a modular multiplication in $\mathbb{F}_p$ is performed in 8 cycles with a throughput of one product per cycle.

4.3.3 Inversion

Fermat's little theorem can be utilized to compute the multiplicative inverse x^{-1} of an integer $x \in \mathbb{F}_p \backslash \{0\}$. Euler's theorem defines that if x and n are positive coprime integers, then

$$x^{\phi(n)} \equiv 1 \pmod{n}, \tag{4.2}$$

where $\phi(n)$ is the Euler's totient function. Since $\phi(p) = p - 1$ for any prime p, then it follows:

$$x^{p-1} \equiv 1 \pmod{p}. \tag{4.3}$$

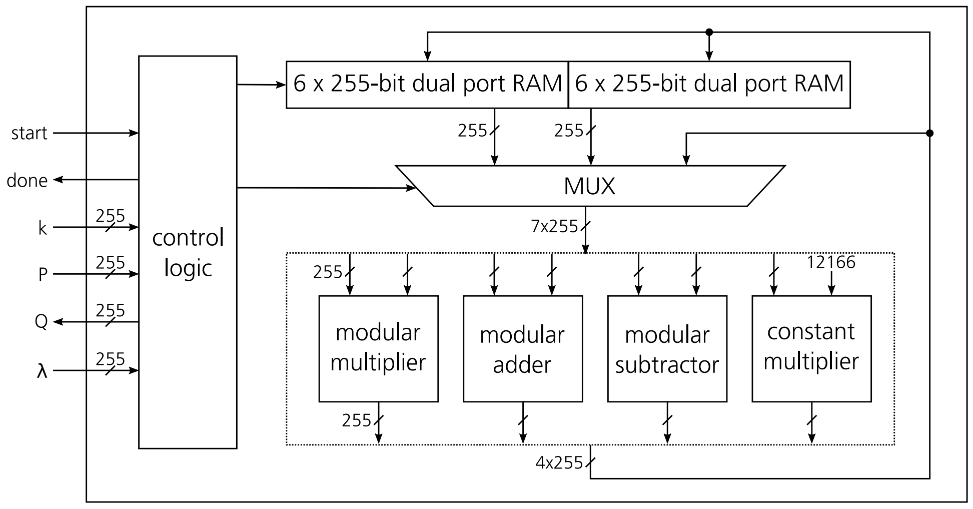


Figure 4.1: X25519 architecture, which contains all control and datapath logic for computing Algorithm 5 (x25519_ladder).

From Equation (4.3), the inverse x^{-1} can be computed as follows:

$$x^{p-2} \equiv x^{-1} \pmod{p}) .$$

Hence, in case of Curve25519, the inversion is given by

$$x^{-1} \equiv x^{2^{255}-21} \pmod{2^{255} - 19} .$$

The exponentiation $x^{2^{255}-21}$ can be computed efficiently with a sequence of squaring and multiplication operations. The logic for the modular inversion is composed by a large FSM that accesses the arithmetic units in a fixed order. We decided to use Fermat's little theorem, instead of the extended euclidean algorithm [54], because it prevents the necessity to instantiate additional arithmetic functions that would require further slices. However, we note that the extend euclidean algorithm can result in a significant speed-up as demonstrated in [49] where only 1,667 clock cycles are required. Compared to this, our modular inversion requires 2,548 cycles.

4.4 Hardware Architecture

We logically divide our design in two parts: the core containing all arithmetic modules including two dedicated 6 × 255-bit memory blocks and the control logic that controls the data flow inside the core. Our implemented architecture is illustrated in Figure 4.1. The control logic consists of a large

FSM that generates the respective control signals for the 255 Montgomery ladder steps and the modular inversion at the end of Montgomery ladder. It is further responsible for the external communication i.e. react on a start signal, process input operands and pull up the done signal once all computations are finished. The core contains four arithmetic modules i.e. modular addition, subtraction, multiplication and constant multiplication. Each arithmetic module features two 255-bit wide input ports that are driven by a multiplexer. The memory blocks are treated similarly and additionally, allow one external 255-bit wide input which is required for setting the initial point and the random value λ. The two memory blocks, which are synthesized as distributed RAM, can hold 6×255-bit operands each that are used to store intermediate values. Our implementation performs one Curve25519 scalar multiplication in 10,465 cycles.

4.4.1 Montgomery Ladder

With respect to Algorithm 5 and the performance of each arithmetic module, the instruction scheduling for one Montgomery ladder step is depicted by Table 4.1. The cycles plotted under the corresponding component (e.g. modular adder A) represent the processing stage. To give an example, t_1 in cycle 1 means that $t_1 = X_1 + Z_1$ is in the first processing stage in the modular adder. In cycle 3, the computation of t_1 is finished and can be further processed by other modules. The control logic implements the corresponding data path and sets the control signals for the respective arithmetic modules. We decided to use a dedicated constant modular multiplier because it allows a noticeable performance improvement while requiring only little additional area resources. The ladder step module hands over the control for every arithmetic unit to the inversion module, once all 255 ladder steps were executed. Our design needs only 7,917 cycles for all 255 ladder steps.

4.5 Results

We synthesized and implemented all modules with Xilinx Vivado 2016.2 and reported values refer to place-and-route designs. Table 4.2 summarizes the number of clock cycles and latency requirements for all X25519 related modules and compares the results to previous FPGA results from Sasdrich and Güneysu [37, 49] and our previous work [26]. Note that compared to our work in [26], we were also able to implement our design on a Zynq-7020. This is enabled by our modular multiplier from Chapter 2 which reduced the required DSP slices. The first thing to note is that the design from Sasdrich

Table 4.1: Instruction scheduling for single X25519 ladder step as described in Algorithm 5 (x25519_ladder) for the modular multiplier (M), the constant modular multiplier (M_c), the adder (A), and the subtractor (Z).

	M		**M_c**		**A**		**Z**	
Cycle	1	9	1	4	1	3	1	3
1	-	-	-	-	t_1	-	t_2	-
2	-	-	-	-	t_3	-	t_4	-
3	t_6	-	-	-	-	t_1	-	t_2
4	t_8	-	-	-	-	t_3	-	t_4
5	t_9	-	-	-	-	-	-	-
6	t_7	-	-	-	-	-	-	-
...	-	-	-	-	-	-	-	-
11	-	t_6	-	-	-	-	-	-
12	-	t_8	-	-	-	-	-	-
13	-	t_9	-	-	t_{10}	-	t_{11}	-
14	X_1	t_7	-	-	-	-	t_5	
15	t_{12}	-	-	-	-	t_{10}	-	t_{11}
16	t_{15}	-	t_{13}	-	-	-	-	t_5
...	-	-	-	-	-	-	-	-
19	-	-	-	t_{13}	t_{14}	-	-	-
...	-	-	-	-	-	-	-	-
21	Z_1	-	-	-	-	t_{14}	-	-
22	-	$\mathbf{X_1}$	-	-	-	-	-	-
23	Z_2	t_{12}	-	-	-	-	-	-
24	X_2	t_{15}	-	-	-	-	-	-
...	-	-	-	-	-	-	-	-
29	-	$\mathbf{Z_1}$	-	-	-	-	-	-
30	-	-	-	-	-	-	-	-
33	-	$\mathbf{Z_2}$	-	-	-	-	-	-
32	-	$\mathbf{X_2}$	-	-	-	-	-	-

Table 4.2: Performance comparison of X25519 implementations in terms of clock cycles and latency requirements.

	This work	**This work**	[**26**]	[**37**]	[**37**]
Platform	Zynq-7030	Zynq-7020	Zynq-7030	Zynq-7020	Zynq-7020
Clock Freq.	115 MHz	84 MHz	115 MHz	100 MHz	200 MHz
Mod. Add.	2	2	2	10	10
Mod. Sub.	2	2	2	10	10
Mod. Mul.	8	8	10	55	55
Mont. Ladder	7,917	7,917	10,711	64,770†	64,770
Mod. Inv.	2,548	2,548	2,928	1,667	14,630
Total	**10,465**	**10,465**	**13,639**	**34,052**	**79,400**
Latency	**92** μs	**125** μs	**118** μs	**340** μs	**397** μs

† Can be operated at 200 MHz.

and Güeneysu can be operated in two clock domains due to huge differences in the frequency requirements of different modules: their Montgomery ladder including the modular arithmetic operates at 200 MHz, while the inversion unit is executed with a maximum frequency of 100 MHZ. Consequently, Sasdrich and Güneysu relate the required 64770 cycles of the Montgomery ladder to the 100 MHz domain i.e. assume that 32885 cycles are needed for it (neglecting the overheads for the domain crossing). Nevertheless, our design, operating in a single domain, achieves increased performance compared to Sasdrich and Güneysu's Montgomery's ladder. For example, our modular multiplication is executed more than five times faster than [37], caused by our speed optimized parallel multiplier. Second, our inversion module can be operated at maximum frequency (it is only limited by the maximum frequency of the multiplier unit). It can be noted that our modular inversion unit using Fermat's Little Theorem appears to be slightly slower in terms of clock cycles than the one presented by Sasdrich and Güneysu, that uses the extended Euclidean algorithm. However, although the extended Euclidean algorithm appears to be faster for hardware based systems, applying Fermat's little theorem to compute the inverse allows higher clock frequencies and lower area requirements, as just the modular multiplication module is reused for it. The overall area utilization for this work, compared with Sasdrich and Güneysu's implementation, is reported in Table 4.3. It can be noted that our design is significantly smaller than the multi-core instantiation, yet larger than the single-core instantiation. Our two 6×255-bit wide memory blocks are synthesized as distributed RAM. In comparison to pre-

Table 4.3: Comparison of area utilization with other X25519 implementations.

Work	FPGA	Cores	Slices	LUTs	Registers	DSP	BRAM
[37]	Zynq-7020	1	1,029	3,592	2,783	20	2
[37]	Zynq-7020	11	11,277	43,875	34,009	220	22
[26]	Zynq-7030	1	8,639	26,483	21,107	260	0
This work	**Zynq-7020**	**1**	**6,161**	**22,627**	**17,924**	**175**	**0**
This work	**Zynq-7030**	**1**	**6,161**	**21,077**	**17,939**	**175**	**0**

vious work [26] we were able to reduce the amount of required DSP blocks, LUTs and registers while achieving lower latency and higher throughput. Also, notice that our design makes inherently use of the randomized projective coordinate countermeasure to thwart DPA. In the extended version [37] of Sasdrich and Güneysu's paper, randomized projective coordinates are also applied, however, their protected design has a penalty of 4110 cycles. In any case, all designs provide a good fit on Xilinx Zynq FPGAs, while leaving enough resources for additional circuits.

4.6 Conclusions

We explored hardware design strategies for X25519 on two Xilinx Zynq FPGAs aimed at low-latency. To reduce the latency, we make use of high-speed arithmetic modules, each carefully optimized to minimize the number of clock cycles as well as the critical path delay, e.g. we use a pipelined 255×255-bit parallel multiplier to perform a modular multiplication in 8 cycles only. Our implementations perform variable-scalar Curve25519 scalar multiplication in 10465 cycles at a maximum frequency of 115 MHz and 84 MHz for the Zynq-7030 and Zynq-7020, respectively. Additionally, randomized projective coordinates were used to counteract side-channel attacks with no area penalty and at the cost of only few clock cycles.

Chapter 5

Hyperelliptic Curve Cryptography

Until today, ECC is a state-of-the art representative of asymmetric cryptography. Targeting a 128-bit security level, numerous speed records for DH key exchange were set by elliptic-curve-based schemes. However, in the past few years, several works based on genus-2 hyperelliptic curves reported promising performance results for several architectures ranging from small microcontrollers [16] to more powerful Intel architectures [19]. This was possible due to the finding of a secure genus-2 curve [18] and its associated Kummer surface, which enables a fast and uniform scalar pseudo-multiplication. Compared to ECC, a genus-2 Kummer surface based key exchange can operate on a field of half the size but features a higher computational complexity. Hyperelliptic curves are in fact a generalization of elliptic curves. However, defining a group structure on hyperelliptic curves of arbitrary genus is more complex. In this chapter, the background on hyperelliptic curve cryptography is presented, which is required for understanding Chapter 6 where a genus-2 key exchange implementation is described.

Outline. Section 5.1 states the general definition of hyperelliptic curves and describes how the Jacobian variety is used to build a group structure. As computations on the Jacobian variety are inefficient, Section 5.2 provides the definition of the associated Kummer surface and its highly efficient addition and doubling formulas.

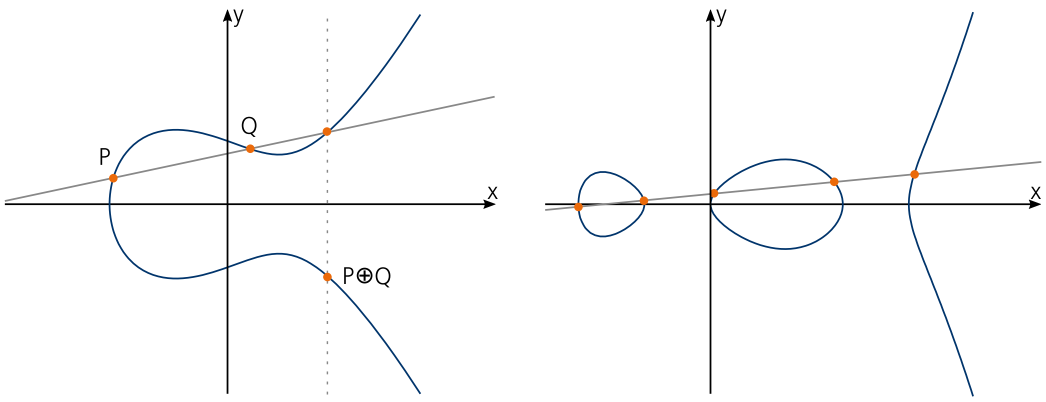


Figure 5.1: Left: Group operation for an elliptic curve using the chord-and-tangent rule. Right: Illustrating how the chord-and-tangent rule is ineffective for a genus-2 hyperelliptic curve.

5.1 Group Law for Hyperelliptic Curves

A hyperelliptic curve of genus g over the field K is given by a curve in the *generalized Weierstrass* equation

$$\mathcal{C} : y^2 + h(x)y = f(x)\,,$$

with the polynomials $f(x), h(x)$, where $f(x)$ is monic[1], and $\deg(f) = 2g + 1$ where g describes the genus. If $\mathrm{char}(K) \neq 2$ then $h(x) = 0$ [39], which is the case for the prime fields $\mathbb{F}_p$ that are in scope of this thesis. A curve of genus $g = 1$ is an elliptic curve, whereas a curve of genus $g > 1$ is a hyperelliptic curve. For example, recall the Curve25519 Equation (4.1) from Section 4.2 that is defined as $y^2 = x^3 + 486662x^2 + x$. As with elliptic curves, the set of rational points for a hyperelliptic curve over $\mathbb{F}_p$ is defined as

$$\mathcal{C}(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p \mid y^2 = f(x)\} \cup \{\infty\}$$

where a point is described as $P = (x,y)$ and its opposite by $\overline{P} = (x, -y)$.

For elliptic curves, a group is formed by the set of points together with the point-at-infinity and the chord-and-tangent rule that serves as the group operation. However, as shown in Figure 5.1 the chord-and-tangent rule does not lead to a group operation for curves of $g > 1$, as a line intersects $\mathcal{C}$ in up to $2g + 1$ points [39].

[1]A polynomial is said to be monic if the leading coefficient (the nonzero coefficient of highest degree) is equal to 1.

5.1.1 The Jacobian Variety

A group structure for a hyperelliptic curve $\mathcal{C}$ of genus g is formed by the *Jacobian variety* or in short the *Jacobian.* The Jacobian is the quotient group

$$J_{\mathcal{C}} = \mathrm{Div}^0_{\mathcal{C}}/\mathrm{Prin}_c$$

where $\mathrm{Div}^0_{\mathcal{C}}$ denotes the *degree-0-divisors* and $\mathrm{Prin}_{\mathcal{C}}$ denotes the *principal divisors.* In the following, the concept of divisors and principal divisors is explained.

Divisors

For a hyperelliptic curve $\mathcal{C}$, various points P can be determined that fulfill the curve equation and are hence located on the curve. This leads to the definition of a divisor D that represents the *formal sum* of points

$$D = \sum_{P \in \mathcal{C}} n_P P, \quad n_P \in \mathbb{Z}\,,$$

where finitely many integers n_P are non-zero. A group is formed by all divisors on $\mathcal{C}$ denoted by $\mathrm{Div}_{\mathcal{C}}$ with the group operation defined by coefficient wise addition [55]. The group operation for two divisors D_1, D_2 can be written as:

$$D_1 \oplus D_2 = \sum_{P \in \mathcal{C}} m_P P + \sum_{P \in \mathcal{C}} n_P P = \sum_{P \in \mathcal{C}} (m_P + n_P) P\,.$$

For example, $(1P_1 + 2P_2) \oplus (1P_1 + 1P_2) = 2P_1 + 3P_2$. However, using this group for cryptographic operations is difficult to implement as it would lead to longer and longer representations of the group elements.

Instead, a further definition is required, which is the degree of a divisor D that is defined as

$$\deg(D) = \sum_{P \in \mathcal{C}} n_P\,.$$

Based on this, the group of *degree-0-divisors* can be defined:

$$\mathrm{Div}^0_{\mathcal{C}} = \{D \in \mathrm{Div}_{\mathcal{C}} : \deg(D) = 0\}\,, \tag{5.1}$$

which is a subgroup of $\mathrm{Div}_{\mathcal{C}}$ [55] and represents the first part of the Jacobian variety.

Principal Divisors

Let a rational function f be an element of the so-called *function field* $\mathbb{F}_p(\mathcal{C})$. An important property of this function field is that all functions have the form $f = \frac{u}{v}$ with the two polynomials u, v. The polynomial u describes the zeroes of f while the polynomial v describes the poles of f. For each function $f \in \mathbb{F}_p(\mathcal{C})$ we associate a principal divisor:

$$\mathrm{div}(f) = \sum_{P \in \mathcal{C}} \nu_P(f) P \,,$$

where $\nu_P(f)$ is a *valuation* function which counts the multiplicity of a zero or pole of a point P:

- $\nu_P(f) = n$ if f has a zero of multiplicity n at P
- $\nu_P(f) = -n$ if f has a pole of multiplicity n at P
- $\nu_P(f) = 0$ otherwise.

A further property of the function field is that $\deg(u) = \deg(v)$, which results in the fact that a principal divisor has degree zero because the number of zeroes equals the number of poles. Therefore, the set of all principal divisors, denoted by $\mathrm{Prin}_{\mathcal{C}}$, is a subgroup of degree-0-divisors and thus the following holds:

$$\mathrm{Prin}_{\mathcal{C}} \leq \mathrm{Div}^0_{\mathcal{C}} \leq \mathrm{Div}_{\mathcal{C}} \,.$$

Finally, the quotient group can effectively be described as $J_{\mathcal{C}} = \mathrm{Div}_{\mathcal{C}}/\mathrm{Prin}_{\mathcal{C}}$.

The Divisor Class Group

The Jacobian variety forms a group where elements are equivalence classes of degree-zero-divisors on $\mathcal{C}$ resulting from functions. It can be shown that each equivalence class contains *semi-reduced divisors* that have the form:

$$D = \sum_{i=1}^{r} P_i - r\infty \,, \qquad P_i \in \mathcal{C} \setminus \{\infty\} \,.$$

An equivalence class contains multiple semi-reduced divisors that are represented by a unique *reduced divisor* with the additional constraint of $r \leq g$. The group operation of the Jacobian is in fact the combination of two reduced divisors. We note that two divisors are called equivalent $D_i \sim D_j$ if they belong to the same equivalence class of $J_{\mathcal{C}}$, which is only the case if $D_i - D_j \in \mathrm{Prin}_{\mathcal{C}}$. This is derived from a standard property of quotient groups.

5.1.2 Addition on the Jacobian

In this section, the addition of two divisors on the Jacobian is described, which is then transformed to an algorithmic description that is known as *Cantor's algorithm.* In the following, we assume a genus $g = 2$ curve over $\mathbb{F}_p$ that has the simplified form

$$\mathcal{C} : y^2 = f(x)\,. \tag{5.2}$$

Let D_1, D_2 be two reduced divisors where $D_1 \neq D_2$ with

$$D_1 = P_1 + P_2 - 2\infty\,, \quad \text{and } D_2 = Q_1 + Q_2 - 2\infty\,.$$

Then there exist exactly one polynomial

$$a(x) = a_0x^3 + a_1x^2 + a_2x + a_3$$

that intersects all four points P_1, P_2, Q_1, Q_2 . By combining Equation (5.2) with $y = a(x)$, we determine the polynomial

$$f(x) - a^2(x) = 0\,,$$

which is a polynomial of degree 6 that intersects the four points P_1, P_2, Q_1, Q_2 and two further points R_1, R_2. This is certainly a principal divisor and can be written as

$$D_3 = P_1 + P_2 + Q_1 + Q_2 + R_1 + R_2 - 6\infty\,.$$

From the previous section we know that two divisors are equivalent if $D_i - D_j \in \mathrm{Prin}_{\mathcal{C}}$ and hence we can infer the following relation

$$D_1 + D_2 = P_1 + P_2 + Q_1 + Q_2 - 4\infty \sim -(R_1 + R_2 - 2\infty)\,. \tag{5.3}$$

Moreover, the divisor

$$P + \overline{P} - 2\infty$$

is a principal divisor as it originates from the function $b(x) = x - a$ which infers that $\overline{P} - \infty \sim -(P - \infty)$. We combine this property with Equation (5.3) and obtain

$$D_1 + D_2 = -(R_1 + R_2 - 2\infty) = \overline{R_1} + \overline{R_2} - 2\infty\,.$$

In fact, this represents the group operation, which is illustrated in Figure 5.2.

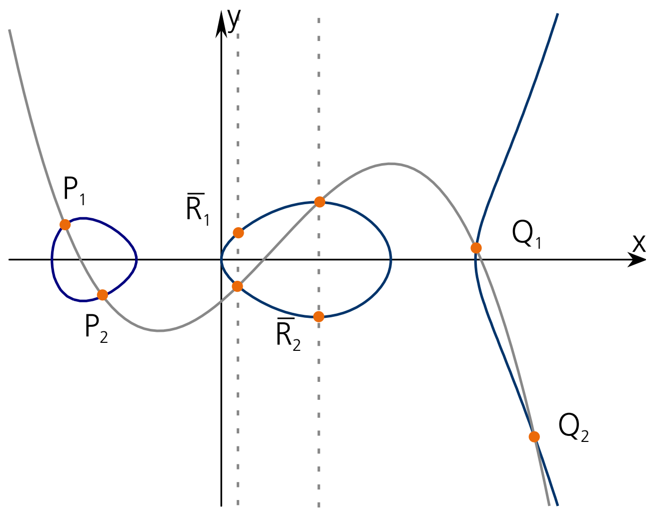


Figure 5.2: Group law for a genus-2 hyperelliptic curve over $\mathbb{R}$.

Mumford Representation

Representing a divisor as the formal sum of points is impractical for implementations. Instead, the Mumford representation can be used which describes a divisor based on polynomials. Each nontrivial divisor class over $\mathbb{F}_p$ can be represented by a unique pair of polynomials $u(x)$ and $v(x)$. Let D be the reduced divisor $D = \sum_i^r P_i - r\infty$ where $P_i \neq \infty, P_i \neq -P_j$ for $i \neq j$ and $r \leq g$. Let $P_i = (x_i, y_i)$, then the *Mumford representation* of $D = \langle u, v \rangle$ can be defined as following

$$u(x) = \prod_{i=1}^{r}(x - x_i)\,,$$

$$\left(\frac{d}{dx}\right)^j [v(x)^2 - f(x)]_{x=x_i} = 0$$

The zeros of $u(x)$ describes the x-coordinates of the points in D, and $v(x)$ is the function that interpolates through all points P_i and in particular $v(x_i) = y_i$. To determine $v(x)$, polynomial interpolation algorithms can be used such as the Lagrange interpolation, i.e. the points $P_1 = (x_1, y_1), ..., P_g = (x_g, y_g)$ correspond to the polynomial:

$$v(x) = \sum_{i=g}^{t} \frac{\prod_{j\neq i}(x - x_j)}{\prod_{j\neq i}(x_i - x_j)} y_i\,.$$

Cantor's Algorithm

Cantor's Algorithm performs a group-wise addition on two reduced divisors. Both input divisors and the output divisor is represented in the Mumford representation. It is implemented by Algorithm 6 (cantor) and can be separated in a composition and a reduction phase. In theory, Cantor's algorithm could

Algorithm 6 cantor: Cantor's algorithm to perform the group operation on two reduced divisors in Mumford representation.

```
Input: Reduced divisors $D_1 = \langle u_1, v_1 \rangle$ and $D_2 = \langle u_2, v_2 \rangle$
Output: Reduced divisor $D \leftarrow D_1 \oplus D_2$
 1: $d_1 \leftarrow \gcd(u_1, u_2) = e_1u_1 + e_2u_2$
 2: $d \leftarrow \gcd(d_1, v_1 + v_2 + h) = c_1d_1 + c_2(v_1 + v_2 + h)$
 3: $s_1 \leftarrow c_1e_1,\ s_2 \leftarrow c_1e_2,\ s_3 \leftarrow c_2$
 4: $u \leftarrow (u_1u_2)/d^2$
 5: $v \leftarrow (s_1u_1v_2 + s_2u_2v_1 + s_3(v_1v_2 + f))/d \mod u$
 6: while $\deg(u) > g$ do                              ▷ reduce the divisor
 7:     $u' \leftarrow (f - vh - v^2)/u$
 8:     $v' \leftarrow (-h - v) \mod u'$
 9:     $u \leftarrow u', v \leftarrow v'$
10: end while
11: return $D = \langle u, v \rangle$
```

be implemented on standard hardware, however, this is inefficient because computing the greatest common divisor (Line 1 and 2), using an algorithm such as the extended Euclidean, is computationally complex.

5.2 Montgomery Arithmetic for Genus-2 Curves over Prime Fields

While Cantor's algorithm works for any curve of genus g, several works were published [56, 57] that determined explicit formulas for genus-2 curves to obtain a faster arithmetic. However, compared to ECC, even those optimized formulas remain computationally inefficient. A different approach was presented by Chudnovsky and Chudnovsky [58] in 1986, in which the authors discussed the application of the scalar multiplication on a Kummer surface associated to a genus-2 hyperelliptic curve. The Kummer surface $\mathcal{K}_\mathcal{C}$ is the image of a rational map κ that identifies the group element $D \in \mathcal{J}_\mathcal{C}$ with its inverse such that $\kappa(D) = \kappa(\overline{D})$. In the elliptic case, the analogue is the projection onto the x-coordinate, i.e. neglecting the y-coordinate, which is a standard approach for increasing the performance. The x-coordinate only arithmetic led to very fast DH key exchange implementations, such as the popular X25519 [12] (see Chapter 4). Even though the mapping to the Kummer surface destroys the group structure, a *pseudo* group operation can still be defined that is sufficient for a DH key exchange.

In the following, an element of the Kummer surface is said to be a point on the Kummer surface. In [58], Chudnovsky and Chudnovsky presented formulas for point doubling $\kappa(2P)$. They also reported the number of field operations for differential point addition i.e. $\kappa(P \oplus Q)$ given $\kappa(P)$, $\kappa(Q)$, and $\kappa(Q \ominus P)$ but did not present the corresponding formulas. The point doubling and differential addition formulas were improved by Gaudry [59] and further refined by Bernstein and Lange [17]. Since those formulas depend on the type and parameterization of the Kummer surface, we state its definition without further explanation.

5.2.1 The Kummer Surface

It is assumed that the genus-2 curve $\mathcal{C}$ is in the so-called *Rosenhain* form

$$\mathcal{C} : y^2 \coloneqq x\,(x-1)\,(x-\lambda)\,(x-\mu)\,(x-\nu)\ .$$

As Gaudry showed [14], the Kummer surface and the genus-2 curve $\mathcal{C}$ are related to each other via the Rosenhain invariants λ, μ, ν:

$$\lambda \coloneqq ac/bd\,, \quad \mu \coloneqq \frac{c\,(1+\sqrt{CD/AB})}{d\,(1-\sqrt{CD/AB})}\,, \quad \nu \coloneqq \frac{a\,(1+\sqrt{CD/AB})}{b\,(1-\sqrt{CD/AB})}\,,$$

where a, b, c, d so-called squared theta constants. Based on the squared theta constants, the dual theta constants A, B, C, D can be determined

$$\begin{aligned} A &\coloneqq a+b+c+d\,, & B &\coloneqq a+b-c-d\,,\\ C &\coloneqq a-b+c-d\,, & D &\coloneqq a-b-c+d\,. \end{aligned}$$

The *fast* Kummer surface $\mathcal{K}_\mathcal{C} \in \mathbb{P}^3$ of [14, 58, 60], is then defined as:

$$\mathcal{K}_\mathcal{C} : Exyzt = \begin{pmatrix} (x^2+y^2+z^2+t^2) \\ -F\,(xt+yz) - G\,(xz+yt) - H\,(xy+zt) \end{pmatrix}^2 ,$$

where

$$F = \frac{a^2-b^2-c^2+d^2}{ad-bc}\,, \quad G = \frac{a^2-b^2+c^2-d^2}{ac-bd}\,, \quad H = \frac{a^2+b^2-c^2-d^2}{ab-cd}\,,$$

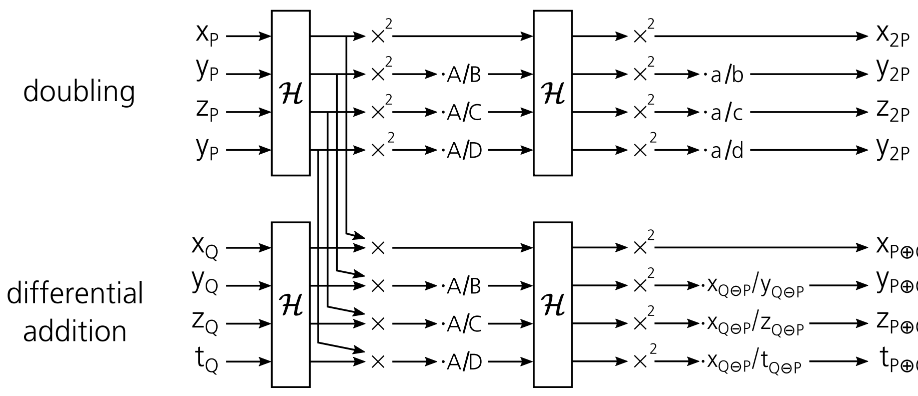


Figure 5.3: Formulas for differential addition and doubling on the fast Kummer surface.

and $E = 4abcd\left(ABCD/\left((ad-bc)(ac-bd)(ab-cd)\right)\right)^2$. For an element $P \in \mathcal{J}_\mathcal{C}$, its image in $\mathcal{K}_\mathcal{C}$ is denoted by

$$\kappa(P) = (x : y : z : t)\,.$$

The identity point $\mathcal{O} = \langle 1, 0\rangle \in \mathcal{J}_\mathcal{C}$, represented in Mumford representation, maps to

$$\kappa(\mathcal{O}) = (a : b : c : d)\ .$$

Fast Kummer Arithmetic

Let $P, Q \in \mathcal{J}_\mathcal{C}$ with $\kappa(P) = (x_P : y_P : z_P : t_P)$, $\kappa(Q) = (x_Q : y_Q : z_Q : t_Q) \in \mathcal{K}_\mathcal{C}$ and assume that the difference $\kappa(Q \ominus P) \in \mathcal{K}_\mathcal{C} = (x_{Q\ominus P} : y_{Q\ominus P} : z_{Q\ominus P} : t_{Q\ominus P})$ is known. Based on [20], Figure 5.3 describes the required field operations for doubling $\kappa(2P)$ and differential addition $\kappa(P \oplus Q)$ with several computations being shared. We note that $\mathcal{H}$ denotes the Hadamard transform which is given by $\mathcal{H} : (x : y : z : t) \mapsto (x_\mathcal{H} : y_\mathcal{H} : z_\mathcal{H} : t_\mathcal{H})$ with

$$x_\mathcal{H} = \overbrace{(x+y)}^{u} + \overbrace{(z+t)}^{v}\,, \qquad z_\mathcal{H} = \overbrace{(x-y)}^{r} + \overbrace{(z-t)}^{s}\,, \tag{5.4}$$

$$y_\mathcal{H} = (x+y) - (z+t)\,, \qquad t_\mathcal{H} = (x-y) - (z-t)\,. \tag{5.5}$$

To further simply the notation, we denote operations in the projective space $\mathbb{P}$ as shown in [16]. First, the multiplication $\mathcal{M}$ that multiplies the corresponding pairs of coordinates from two distinct points in $\mathbb{F}_p$:

$$\mathcal{M} : \left((x_1 : y_1 : z_1 : t_1), (x_2 : y_2 : z_2 : t_2)\right) \mapsto (x_1x_2 : y_1y_2 : z_1z_2 : t_1t_2)\ .$$

And second, the special case where the two points are equal, i.e. squaring in $\mathbb{F}_p$ the corresponding pairs of coordinates:

$$\mathcal{S} : (x : y : z : t) \mapsto \left(x^2 : y^2 : z^2 : t^2\right) .$$

By sharing the intermediate values, as illustrated in Figure 5.3, an efficient algorithm for a combined double-and-add can be determined. Algorithm 7 (dbladd) denotes this procedure requiring only $7\mathrm{M} + 12\mathrm{S} + 9\mathrm{M}_c$ field operations, where M stands for modular multiplication, S for modular squaring, and M_c for constant multiplication. As we show in the next section, an efficient routine for scalar multiplication can then be constructed using the combined double-and-add and the Montgomery ladder from Section 3.4 Algorithm 4 (mont_ladder).

Algorithm 7 dbladd: Combined double-and-add on a Kummer surface of a genus-2 curve.

```
Input: $\kappa(P), \kappa(Q),\ \kappa(Q \ominus P) \in \mathcal{K}_\mathcal{C}$
Output: $\kappa(2P),\ \kappa(P \oplus Q) \in \mathcal{K}_\mathcal{C}$
 1: $V_5 \leftarrow \kappa(P)$
 2: $V_6 \leftarrow \kappa(Q)$
 3: $V_7 \leftarrow \left(\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}\right)$
 4: $V_8 \leftarrow \left(\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d}\right)$
 5: $V_9 \leftarrow \left(1 : \frac{x_{Q\ominus P}}{y_{Q\ominus P}} : \frac{x_{Q\ominus P}}{z_{Q\ominus P}} : \frac{x_{Q\ominus P}}{t_{Q\ominus P}}\right)$
 6: $(V_1, V_2) \leftarrow (\mathcal{H}(V_5), \mathcal{H}(V_6))$
 7: $(V_3, V_4) \leftarrow (\mathcal{S}(V_1), \mathcal{M}(V_1, V_2))$
 8: $(V_5, V_6) \leftarrow (\mathcal{M}(V_3, V_7), \mathcal{M}(V_4, V_7))$
 9: $(V_1, V_2) \leftarrow (\mathcal{H}(V_5), \mathcal{H}(V_6))$
10: $(V_3, V_4) \leftarrow (\mathcal{S}(V_1), \mathcal{S}(V_2))$
11: $(V_5, V_6) \leftarrow (\mathcal{M}(V_3, V_8), \mathcal{M}(V_4, V_9))$
12: return $\kappa(2P) = V_5,\ \kappa(P \oplus Q) = V_6$
```

Chapter 6

Kummer Surface Based DH Key Exchange on an FPGA

In this chapter, we present two hardware implementations of the DH key exchange based on the Kummer surface of Gaudry and Schost's genus-2 curve targeting a 128-bit security level. We describe a single-core architecture for low-latency applications and a multi-core architecture for high-throughput applications. Parts of this chapter have been published in [28].

Outline. In Section 6.1 we summarize the motivation of this work and the main design decisions that enabled our high-speed design. Section 6.2 describes Gaudry and Schost's hyperelliptic curve and its Kummer surface, and summarize the scalar multiplication on this Kummer surface using the Montgomery ladder in Section 6.3. In Section 6.4, a description of the single-core and multi-core hardware architectures is provided including a performance analysis and comparison to related work. Finally, we conclude and discuss the results in Section 6.5.

6.1 Introduction

In 2006, Bernstein and Lange [17] showed in a cost analysis for software that a genus-2 based implementation is potentially 1.5-times faster than a comparable elliptic curve based implementation. At that time, however, a secure Kummer surface of a genus-2 curve was not found yet. Since genus-2 point counting is computationally expensive, it took further six years until Gaudry and Schost [18] presented a twist-secure Kummer surface targeting a 128-bit security level. So far, investigations of the DH key exchange on the Kummer surface of genus-2 curves were confined to software implementations [16, 19, 20]. While these software implementations already showed the performance advantages of genus-2 curves, the design of efficient hardware is a fundamentally different task. Best performance results are only obtained when each module is carefully optimized with optimally matched timing characteristics to one another. In this work, we show that the Kummer surface of Gaudry and Schost's genus-2 curve can be used to perform very fast DH key exchanges in hardware. The main design decisions that influenced our results are described in Section 6.4 and summarized below:

Interleaving two scalar multiplications. Due to the serial nature of the considered ladder, multiple hardware modules, (such as the modular multiplier), operate below full capacity. This allows for a second scalar multiplication to be efficiently interleaved by carefully scheduling the required field operations. The obtained instruction schedule leaves the number of cycles unaltered while effectively doubling the throughput. Note that this interleaved scalar multiplication can also be used as a countermeasure against fault attacks by performing both scalar multiplications on the same input point and check the results for equivalence.

Efficient representation of constant values. For improved performance, we instantiate a dedicated circuit for multiplying field elements with 12-bit constants in each ladder step. Compared to a conventional modular multiplication, the constant modular multiplier requires only 4 clock cycles instead of 7. Some constants, however, are negative; the naive approach would be to convert them to positive elements of the prime field and then use the modular multiplier for multiplication. In order to avoid the increased memory requirements and decreased performance of this naive approach, we neglect the sign when storing the constants and include the conditional negation logic inside the constant modular multiplier.

High-speed modular multiplier. The performance of the scalar multiplication is strongly correlated with the performance of the modular multiplier. We reuse the multiplier presented in Chapter 2, which is explicitly optimized for Mersenne prime fields, and modify it by applying the non-standard tiling technique [2] to further improve its performance. In this way, we also reduce the number of required DSP blocks by 10%.

Algorithm 8 scalar_mult: Unwrap input point to Montgomery ladder on $\mathcal{K}_{\mathcal{C}}$ followed by point wrapping. It is assumed that the public key (respectively public generator) is in 381-bit wrapped representation.

Input: $\left(k = \sum_{i=0}^{250} k_i 2^i\right) \in [0, 2^{251}), \underline{\kappa(P)}$ for $\kappa(P)$ in $\mathcal{K}_{\mathcal{C}}$.
Output: $\underline{\kappa(Q)}$ for $\kappa(Q) \leftarrow \kappa([k]P)$ in $\mathcal{K}_{\mathcal{C}}$.

```
1: $\kappa(P) \leftarrow \mathsf{unwrap}\left(\underline{\kappa(P)}\right)$    $\triangleright$ compute 4-tuple representation of $\kappa(P)$
2: $\kappa(Q) \leftarrow \mathsf{mont\_ladder\_kummer}\left(k, \kappa(P), \underline{\kappa(P)}\right)$
3: $\underline{\kappa(Q)} \leftarrow \mathsf{wrap}\left(\kappa(Q)\right)$  $\triangleright$ compute wrapped 381-bit representation of $\kappa(Q)$
4: return $\underline{\kappa(Q)}$
```

6.2 Preliminaries

Our implemented DH key exchange works the same as the one described by Renes et al. [16]. A detailed description of the underlying theory can be found in Chapter 5. A point $\kappa(P)$ is represented by a 4-tuple where each element is 127-bit wide which sums up to 508 bit in total. As described in [16, 60], we assume that the public key (respectively public generator) is represented by a 3-tuple in its wrapped 381-bit representation denoted by $\underline{\kappa(P)}$. Renes et al. [16] showed that keeping the input points in their wrapped representation offers two advantages: first, it reduces the required amount of data that needs to be transmitted and second, it results in a speed-up for the ladder computation.

For an ephemeral key exchange, the scalar multiplication is performed twice: once for computing an entity's public key, where the public generator is the input point, and once for computing a shared secret, where the other entity's public-key is the input point.

Key exchange. *Let $\underline{\kappa(P)}$ be the public generator (respectively public key) in its wrapped representation and k be the 251-bit secret key. We then compute $\kappa(Q) \leftarrow \kappa([k]P)$ and derive the generated public key (respectively the shared secret) as $\underline{\kappa(Q)}$.*

The scalar multiplication is implemented by Algorithm 8 (scalar_mult) and uses three functions: unwrap computes the 4-tuple representation of the input point, mont_ladder_kummer multiplies the unwrapped input point by a scalar value using the Montgomery ladder [4], and wrap finally computes the 381-bit wrapped representation of the output point; all these functions are described in detail in Section 6.3. In the previous chapter, we stated the general definition of the Kummer surface. In our implementation we use the

Kummer surface of Gaudry and Schost's genus-2 hyperelliptic curve [18], and thus we firstly summarize the definition of this curve.

6.2.1 Gaudry and Schost's Genus-2 Hyperelliptic Curve

The genus-2 hyperelliptic curve $\mathcal{C}$ of Gaudry and Schost [18] is defined over the prime field $\mathbb{F}_p$ with $p = 2^{127} - 1$. From the previous chapter, recall the Rosenhain model of the curve $\mathcal{C}$, which can be written as follows:

$$\mathcal{C} : \, g^2 := x\,(x-1)\,(x-\lambda)\,(x-\mu)\,(x-\nu) \ ,$$

where the Rosenhain invariants are defined as

$$\begin{aligned}
\lambda &:= ac/bd = \texttt{0x15555555555555555555555555555552}\,,\\
\mu &:= ce/df = \texttt{0x73E334FBB315130E05A505C31919A746}\,,\\
\nu &:= ae/bf = \texttt{0x552AB1B63BF799716B5806482D2D21F3}\,,
\end{aligned}$$

the squared theta constants are set to

$$\begin{gathered}
a = -11\,, \quad b = 22\,, \quad c = 19\,, \quad \text{and} \quad d = 3\,,\\
e/f = (1 + \sqrt{CD/AB})/(1 - \sqrt{CD/AB})\,,
\end{gathered}$$

and the dual theta constants are set to

$$\begin{aligned}
A &:= a + b + c + d = 33\,, & B &:= a + b - c - d = -11\,,\\
C &:= a - b + c - d = -17\,, & D &:= a - b - c + d = -49\,.
\end{aligned}$$

6.3 Compression and Scalar Multiplication

As described in Algorithm 8 (scalar_mult), we assume that the input and output points are in their wrapped representation. The wrapped representation of the point $\kappa(P) = (x : y : z : t)$ in $\mathcal{K}_\mathcal{C}$ is composed of a 3-tuple and denoted by $\underline{\kappa(P)} = (x/y, x/z, x/t)$. Algorithm 9 (unwrap) implements the point unwrapping, which consists of 4 multiplications in $\mathbb{F}_p$. The wrapping function is described in Algorithm 10 (wrap); it consists of a finite field inversion and 7 multiplications. Algorithm 11 (mont_ladder_kummer) describes the Montgomery ladder for the scalar multiplication on the Kummer surface of Gaudry and Schost's genus-2 curve. The constants that are stored in V_7

Algorithm 9 unwrap: $(x/y, x/z, x/t) \mapsto (x : y : z : t)$ Unwrap point to its 508-bit representation.

```
Input: $(x/y, x/z, x/t)$.
Output: $(x : y : z : t)$.
1: $(V_1, V_2, V_3) \leftarrow ((x/z)(x/t), (x/y)(x/t), (x/y)(x/z))$
2: $V_4 \leftarrow V_3 (x/t)$
3: return $(V_4 : V_1 : V_2 : V_3)$
```

Algorithm 10 wrap: $(x : y : z : t) \mapsto (x/y, x/z, x/t)$ Compute wrapped 381-bit representation.

```
Input: $(x : y : z : t)$.
Output: $(x/y, x/z, x/t)$.
1: $V_1 \leftarrow yz$
2: $V_2 \leftarrow x/(V_1 t)$                                    ▷ inversion
3: $V_3 \leftarrow V_2 t$
4: return $(V_3 z, V_3 y, V_1 V_2)$
```

and V_8 are projectively derived from the squared theta constants (a, b, c, d) and the dual theta constants (A, B, C, D) respectively (see Section 6.2.1):

$$\left(\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d}\right) = (114 : -57 : -66 : -418) \ ,$$
$$\left(\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}\right) = (-833 : 2499 : 1617 : 561) \ .$$

The Montgomery ladder consists of 251 ladder steps, each one performing a differential addition and a doubling operation. Each ladder step includes a conditional swap of two pairs of coordinates.

6.4 Hardware Architectures

The implementation of Algorithm 8 (scalar_mult) is the essential task of our hardware design. We present a single-core architecture for low-latency applications and a multi-core architecture for high-throughput applications. Our single-core architecture performs two scalar multiplications on the Kummer surface at a time by scheduling the field operations for point addition and point doubling such that it is possible to interleave a second scalar multiplication with no cycle penalty. The top-view architecture is illustrated in Figure 6.1. It takes two points in their wrapped representation as input,

Algorithm 11 mont_ladder_kummer: Montgomery ladder using combined differential double-and-add.

```
Input: $\left(k = \sum_{i=0}^{250} k_i 2^i\right) \in [2^{250}, 2^{251})$, $\left(\kappa(P), \underline{\kappa(P)}\right) \in \mathcal{K}_{\mathcal{C}}^2$.
Output: $\kappa(Q) = (x_Q : y_Q : z_Q : t_Q)$ for $\kappa(Q) \leftarrow \kappa([k]P)$ in $\mathcal{K}_{\mathcal{C}}$.
 1: $V_5 \leftarrow (a : b : c : d)$                                  ▷ cf. Algorithm 4, $R_1 \leftarrow \mathcal{O}$
 2: $V_6 \leftarrow (x_P : y_P : z_P : t_P)$                          ▷ cf. Algorithm 4, $R_2 \leftarrow P$
 3: $V_7 \leftarrow \left(\frac{1}{A} : \frac{1}{B} : \frac{1}{C} : \frac{1}{D}\right)$
 4: $V_8 \leftarrow \left(\frac{1}{a} : \frac{1}{b} : \frac{1}{c} : \frac{1}{d}\right)$
 5: $V_9 \leftarrow \left(1 : \frac{x_P}{y_P} : \frac{x_P}{z_P} : \frac{x_P}{t_P}\right)$              ▷ representation of $\underline{\kappa(P)}$
 6: for $i = |k| - 1$ downto $0$ do
 7:     $(V_1, V_2) \leftarrow \mathsf{cswap}\,(k_i \oplus k_{i+1}, (V_5, V_6))$               ▷ $s_{251} = 0$
 8:     $(V_1, V_2) \leftarrow (\mathcal{H}(V_1), \mathcal{H}(V_2))$
 9:     $(V_3, V_4) \leftarrow (\mathcal{S}(V_1), \mathcal{M}(V_1, V_2))$
10:     $(V_5, V_6) \leftarrow (\mathcal{M}(V_3, V_7), \mathcal{M}(V_4, V_7))$
11:     $(V_1, V_2) \leftarrow (\mathcal{H}(V_5), \mathcal{H}(V_6))$
12:     $(V_3, V_4) \leftarrow (\mathcal{S}(V_1), \mathcal{S}(V_2))$
13:     $(V_5, V_6) \leftarrow (\mathcal{M}(V_3, V_8), \mathcal{M}(V_4, V_9))$
14: end for
15: $(V_1, V_2) \leftarrow \mathsf{cswap}\,(k_0, (V_5, V_6))$
16: return $\kappa(Q) = V_2$
```

processes them, and returns two points in their wrapped representation as output. We logically divide our single-core design into three parts that are described in the next subsections: memory, datapath, and control logic. Further we describe a multi-core architecture that instantiates 4 independently operating cores and can perform up to 8 scalar multiplications with different keys and input points.

Note that the two interleaved scalar multiplications can be inherently used as a redundancy countermeasure to thwart fault attacks in our designs, i.e. by performing two interleaved scalar multiplications on the same points with the same key and then check the result for equivalence. This countermeasure can be applied to both our single- and multi-core architectures without applying any changes to the presented hardware designs.

6.4.1 Memory

The memory consists of a 16 × 127-bit register file and a 6 × 127-bit simple dual-port RAM. The register file is divided in four larger blocks, where each block is 4 × 127-bit wide. We follow the logical structure of Algorithm 11

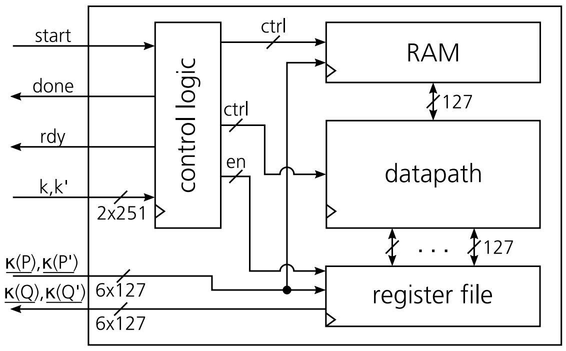


Figure 6.1: Single-core architecture, which contains all control and datapath logic for computing Algorithm 8 (scalar_mult).

(mont_ladder_kummer) in which operations are performed on two points at a time (e.g. V_1, V_2 on line 8). We also use a simple dual-port RAM for storing the wrapped input point $\frac{x_p}{y_p}$, $\frac{x_p}{z_p}$, and $\frac{x_p}{t_p}$, which is accessed in read-only mode. Note that when no design constraints are set, the used synthesis tool instantiates distributed RAM instead of block RAM for storing this point. We found out that forcing the synthesis tool to use block RAM resulted in a 10% decrease of the maximum clock frequency.

6.4.2 Datapath

The datapath including the register file is shown in Figure 6.2. It implements the required field operations in $\mathbb{F}_p$. The register blocks R_i and R'_i for $i \in [1, 2]$ are required for storing intermediate values of the first and the second scalar multiplication, respectively. The register blocks R_1 and R'_1 are initialized with the constants $V_5 = (a : b : c : d)$ whenever Algorithm 8 (scalar_mult) is started. The modular multiplier is preceded by the multiplexer m_3 that allows to perform field operations using various input sources. The output of the constant modular multiplier and the Hadamard module serve as fast forward input paths for the modular multiplier. These fast forward paths are required when data needs to be processed immediately without any further delay. Moreover, the modular multiplier can process 127-bit inputs that originate from the RAM and are required in each ladder step (e.g. multiplication by $\frac{x_p}{y_p}$). We can store each field operation output in the register blocks, i.e. R_i and R'_i, by accordingly selecting the signals with the multiplexers m_1 and m_2. Although large multiplexers result in an increased area utilization, they allow greater flexibility in scheduling instructions which leads to higher

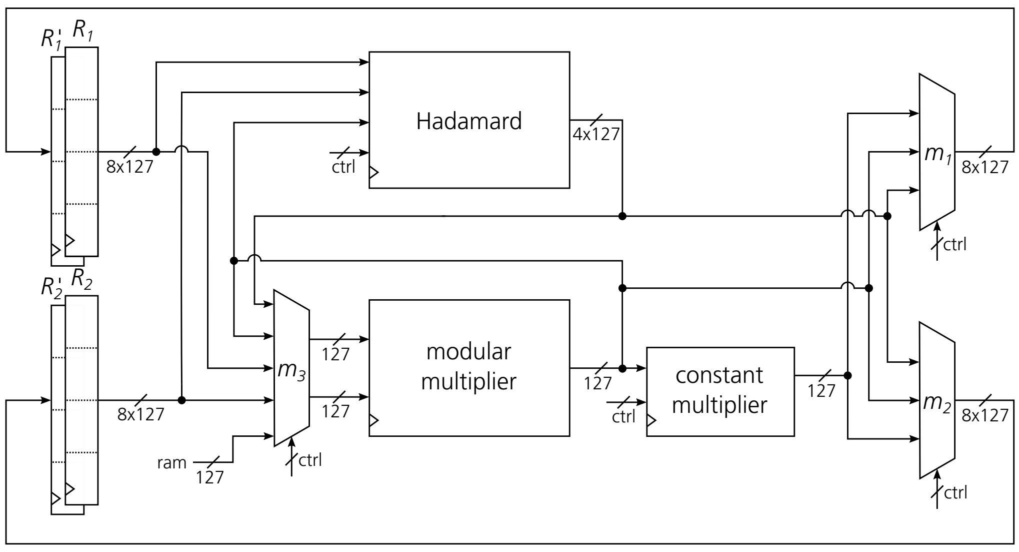


Figure 6.2: Datapath including register file.

overall performance. All select and enable signals in Figure 6.2 are driven by the control logic (see Section 6.4.3).

Modular Multiplier

We reuse the modular multiplier design from Chapter 2, but further extend it by applying the non-standard tiling technique [2]. Our multiplier returns the result after 7 cycles including the reduction step. This property is not only beneficial for the performance, but also required in order to interleave a second scalar multiplication. Our implemented modular multiplier is used for both squaring and multiplication in $\mathbb{F}_p$. Figure 6.3 shows the hardware architecture of our modular multiplier.

In modern FPGAs, DSP blocks typically contain asymmetric multipliers, e.g. in case of the Zynq-7020 FPGA a 17×24-bit multiplier is contained in each DSP block. In order to exploit these asymmetries to reduce the amount of DSP blocks used to perform large multiplications, different optimization strategies were proposed [1, 61, 62]. In particular, the authors of [61] showed that operand decomposition boils down to a tiling problem, where each tile represents the result of a smaller digit-product computation. Roy et al. [2] proposed the non-standard tiling algorithm as a solution to this tiling problem. They presented a formal procedure to compute this non-standard tiling for large multipliers with arbitrary operand sizes. The goal is to determine a tiling configuration that covers the 127-bit multiplier while instantiating as

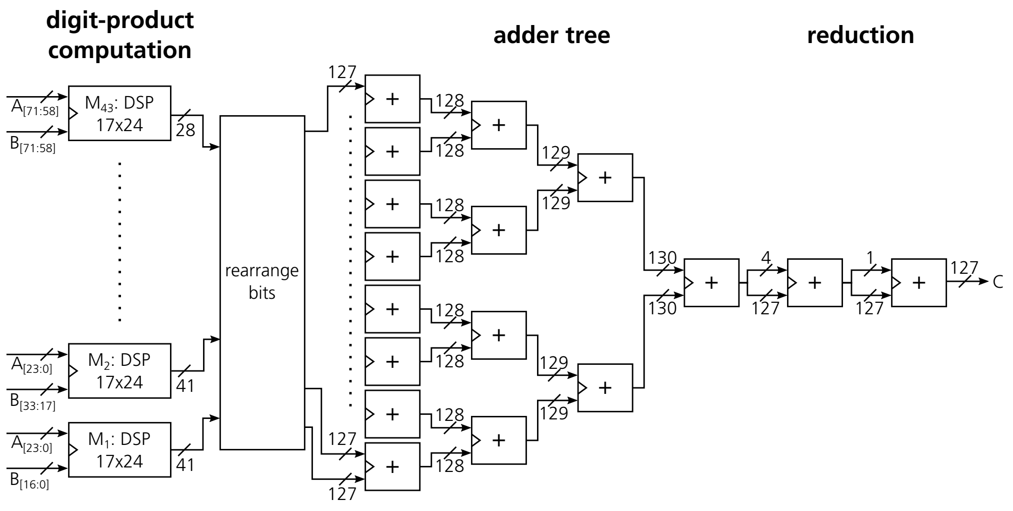


Figure 6.3: Architecture of the modular multiplier, as similarly shown in Chapter 2.

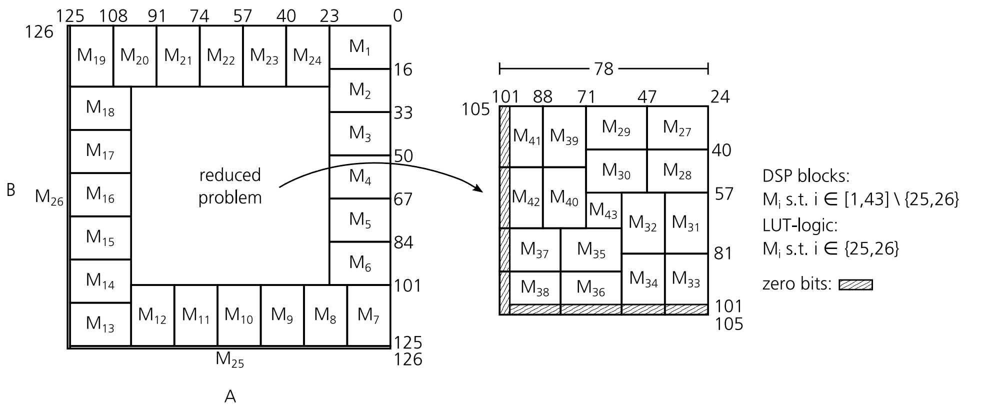


Figure 6.4: Left: Non-standard tiling [2] for 127×127-bit multiplier. Right: Non-standard tiling for smaller 78×78-bit multiplier.

few tiles as possible. For a 127×127-bit multiplier, Figure 6.4 presents the implemented non-standard tiling [2]. The horizontal side represents operand A and the vertical side represents operand B. The size of the tiles M_i where $i \in [1, 43] \setminus \{25, 26\}$ corresponds to the asymmetric multiplier widths and can consequently be implemented in a single DSP block. The two tiles M_{25} and M_{26}, however, correspond to a 126×1-bit multiplier and a 1×127-bit multiplier, respectively, both implemented in LUT logic. With this initial tiling, the problem of finding an efficient placement for a 127×127-bit multiplier is reduced to a 78×78-bit multiplier. Again, we perform non-standard tiling for the reduced problem which results in a smaller 14×14-bit multiplier M_{43}. Comparing non-standard-tiling with standard-tiling, only 41 DSP blocks are required instead of 64 [1].

Constant Modular Multiplier

In order to speed up the Montgomery ladder, we instantiate a constant modular multiplier that multiplies one of the constants in $\left\{\frac{1}{a}, \frac{1}{b}, \frac{1}{c}, \frac{1}{d}, \frac{1}{A}, \frac{1}{B}, \frac{1}{C}, \frac{1}{D}\right\}$ with a variable 127-bit operand. The constant modular multiplier returns with a latency of 4 cycles, which is 3 cycles less than the generic modular multiplier. The variable 127-bit operand can be broken down into $6 = \lceil 127/24 \rceil$ tiles. Since each constant is less than 17-bit, the constant modular multiplier can be implemented with only six 17×24 DSP blocks and some LUTs for the adder tree. The multiplication itself is pipelined and followed by two reduction steps including a conditional negation. The conditional negation is required for the multiplication with projectively negative constants, i.e. $\frac{1}{b}, \frac{1}{c}, \frac{1}{d}$, and $\frac{1}{A}$. For all other constants, i.e. $\frac{1}{a}, \frac{1}{B}, \frac{1}{C}$, and $\frac{1}{D}$, the negation output is ignored. All constants are hard-decoded and then selected for multiplication via a select signal. Overall, 12 modular multiplications in each ladder step can be replaced by constant multiplications.

Hadamard Transform

A further operation in Algorithm 11 (mont_ladder_kummer) is the Hadamard transform. It is essentially composed of 4 modular additions and 4 modular subtractions, which we implemented using 2 modular adders and 2 modular subtractors. In order to parallelize the execution of independent operations, a modular adder is implemented using two addition circuits that are connected in series, each one having a clocked register output. The first adds two 127-bit wide operands and the second reduces the sum again by using Crandall's fast reduction [30]. Because a register is placed after each addition circuit, a result is obtained each cycle after an initial delay of 2 cycles. The modular

Table 6.1: Instruction scheduling for two successive Hadarmard computations as in line 8 of Algorithm 11 (mont_ladder_kummer) using modular addition (A) and subtraction (Z).

	$\mathbf{A_1}$		$\mathbf{A_2}$		$\mathbf{Z_2}$		$\mathbf{Z_2}$	
Cycle	1	3	1	3	1	3	1	3
1	u_1	-	v_1	-	r_1	-	s_1	-
2	u_2	-	v_2	-	r_2	-	s_2	-
3	$x_{\mathcal{H}_1}$	u_1	$z_{\mathcal{H}_1}$	v_1	$y_{\mathcal{H}_1}$	r_1	$t_{\mathcal{H}_1}$	s_1
4	$x_{\mathcal{H}_2}$	u_2	$z_{\mathcal{H}_2}$	v_2	$y_{\mathcal{H}_2}$	r_2	$t_{\mathcal{H}_2}$	s_2
5	-	$\mathbf{x_{\mathcal{H}_1}}$	-	$\mathbf{z_{\mathcal{H}_1}}$	-	$\mathbf{y_{\mathcal{H}_1}}$	-	$\mathbf{t_{\mathcal{H}_1}}$
6	-	$\mathbf{x_{\mathcal{H}_2}}$	-	$\mathbf{z_{\mathcal{H}_2}}$	-	$\mathbf{y_{\mathcal{H}_2}}$	-	$\mathbf{t_{\mathcal{H}_2}}$

subtraction circuit is implemented similarly; modular addition and modular subtraction are both implemented in LUT logic.

Two successive Hadamard transforms, i.e. $\mathcal{H}(V_1), \mathcal{H}(V_2)$, are computed at the beginning of each ladder step before any other computation can take place. Therefore, the modular adder and the modular subtractor circuits are connected with a multiplexer in a way that two Hadamard transforms are finished in successive clock cycles. Table 6.1 shows the scheduling for a Hadamard transform of two points, i.e. $V_1 = (x_1 : y_1 : z_1 : t_1)$ and $V_2 = (x_2 : y_2 : z_2 : t_2)$, plotted over cycles to compute Equation (5.4) and Equation (5.5) (see Section 6.3). The cycles plotted under the corresponding component (e.g. modular adder A_1) represent the processing stage. To give an example, u_1 in cycle 1 means that $u_1 = x_1 + y_1$ is in the first processing stage in the modular adder. In cycle 3, the computation of u_1 is finished and can be further processed by other modules. The transformed points $\mathcal{H}(V_1)$ and $\mathcal{H}(V_2)$ are returned in the 5th cycle and in the 6th cycle, respectively.

To reduce the number of modular reductions and hence the number of required cycles, lazy reduction is a popular technique. In software, lazy reduction comes typically for free because field elements are often smaller than a multiple of the word size which results in unused bits at higher positions. In hardware, however, lazy reduction leads to increased memory requirements, larger multipliers, and a more complex control logic to distinguish between reduced and unreduced field elements when initiating a modular multiplication. Therefore, lazy reduction was not applied here.

Table 6.2: Latency in cycles and throughput in operations per cycles of field operations.

Operation	**Latency (cycles)**	**Throughput (op/cycles)**
Addition/subtraction in $\mathbb{F}_p$	2	1
Multiplication/squaring in $\mathbb{F}_p$	7	1
Constant multiplication in $\mathbb{F}_p$	4	1
Inversion in $\mathbb{F}_p$	952	1/476
Hadamard transform	4	1/2

6.4.3 Control Logic

The control logic takes care of performing the necessary memory operations in the register file and RAM, and schedules the instructions required by Algorithm 8 (scalar_mult). The unwrapping and wrapping function, and the Montgomery ladder logically divide the control logic into separate control blocks. The control logic is implemented using a Finite State Machine (FSM). Inside the FSM multiple counters are used to track the processing status of arithmetic modules such as the modular multiplier. For an efficient instruction scheduling, the latency and throughput characteristics of the underlying functions such as modular multiplication and Hadamard transform are required. Table 6.2 shows the performance of the field operations in $\mathbb{F}_p$ including Hadamard transform, which is composed of modular additions and subtractions. The throughput denotes how often an instruction can be scheduled, e.g. a throughput of 1/2 (op/cycles) means 1 instruction can be scheduled in 2 cycles. Table 6.3 reports the latency of all high-level operations. The Montgomery ladder comprises 90 percent of all cycles, and hence it is crucial to efficiently schedule field-level instructions.

Montgomery Ladder

Table 6.4 shows the instruction scheduling for a Montgomery ladder step for two scalar multiplications. Instructions of the second scalar multiplication are complemented by a prime symbol, e.g. y_1'. Overall, 251 Montgomery ladder steps are executed, each implementing a combined differential double-and-add, which takes 41 cycles to run. All scheduled instructions denote the expected output, e.g. in cycle 5 the squaring y_3 is an abbreviation and

Table 6.3: Latency in terms of cycle count (CC) of high-level functions.

Operation	**Latency (CC)**
Unwrap	30
Combined differential double-and-add	41
Montgomery ladder	10,302
Wrap	998
Scalar multiplication	11,330

stands for the computation of $y_3 = V_{3,y} = V_{1,y}V_{1,y}$ as described in line 9 of Algorithm 11 (mont_ladder_kummer). The conditional-swap function is implemented with no timing-penalty by simply swapping the arguments of the first two Hadamard transforms. Our control logic schedules modular multiplications and multiplications by constants in parallel for best performance results. Note that the constant multiplier uses the direct output of the modular multiplier.

Modular Inversion

We use Fermat's little theorem to compute the multiplicative inverse x^{-1} of an integer $x \in \mathbb{F}_p \backslash \{0\}$. The finite field inversion is given by $x^{-1} \equiv x^{2^{127}-3}$.This exponentiation is computed with a sequence of 126 modular squarings and 10 modular multiplications as described by Renes et al. [16]. Due to the serial nature of the modular inversion, there is little room for scheduling operations of a single inversion in parallel. This, however, enables us to schedule a second independent modular inversion in parallel by repeating each operation for the corresponding operands with a one cycle delay.

6.4.4 Multi-Core Architecture

For multi-core architectures, the amount of cores which can be instantiated in parallel is strongly limited by the number of DSP blocks available on the target FPGA device. Our multi-core architecture implements 4 independently operating single-cores each featuring its own control logic. As a result, up to 8 scalar multiplications with different keys and input points can be computed.

Table 6.4: Instruction scheduling for single ladder step as described in Algorithm 11 (mont_ladder_kummer) for the modular multiplier (M), the constant modular multiplier (M_c), and the Hadamard transform module (H).

	M		**H**		$\mathbf{M_c}$			**M**		**H**		$\mathbf{M_c}$	
Cycle	1	8	1	5	1	5	**Cycle**	1	8	1	5	1	4
1	-	-	$\mathcal{H}_1$	-	-	-	28	z_3	z_3'	-	-	z_5'	y_6'
2	-	-	$\mathcal{H}_2$	-	-	-	29	t_3	t_3'	-	-	t_5'	z_6'
...	-	-	-	-	-	-	30	x_3	x_3'	-	-	x_5'	t_6'
5	y_3	-	-	$\mathcal{H}_1$	-	-	31	y_6	y_4	$\mathcal{H}_2'$	-	-	x_6'
6	y_4	-	-	$\mathcal{H}_2$	-	-	32	z_6	z_4	-	-	-	z_5'
7	z_4	-	-	-	-	-	33	t_6	t_4	-	-	-	t_5'
8	t_4	-	-	-	-	-	34	x_4	y_3	$\mathcal{H}_1'$	-	y_5	x_5'
9	x_4	-	-	-	-	-	35	y_4'	z_3	-	$\mathcal{H}_2'$	z_5	-
10	z_3	-	-	-	-	-	36	z_4'	t_3	-	-	t_5	-
11	t_3	-	-	-	-	-	37	t_4'	x_3	-	-	x_5	-
12	x_3	y_3	$\mathcal{H}_1'$	-	y_5	-	38	y_3'	$\mathbf{y_6}$	-	$\mathcal{H}_1'$	-	$\mathbf{y_5}$
13	-	y_4	$\mathcal{H}_2'$	-	y_6	-	39	z_3'	$\mathbf{z_6}$	-	-	-	$\mathbf{z_5}$
14	-	z_4	-	-	z_6	-	40	t_3'	$\mathbf{t_6}$	-	-	-	$\mathbf{t_5}$
15	-	t_4	-	-	t_6	-	41	x_3'	$\mathbf{x_4}$	-	-	-	$\mathbf{x_5}$
16	y_3'	x_4	-	$\mathcal{H}_1'$	x_6	y_5	1	y_6'	y_4'	-	-	-	-
17	y_4'	z_3	-	$\mathcal{H}_2'$	z_5	y_6	2	z_6'	z_4'	-	-	-	-
18	z_4'	t_3	-	-	t_5	z_6	3	t_6'	t_4'	-	-	-	-
19	t_4'	x_3	-	-	x_5	t_6	4	x_4'	y_3'	-	-	y_5'	-
20	x_4'	-	$\mathcal{H}_2$	-	-	x_6	5	-	z_3'	-	-	z_5'	-
21	z_3'	-	-	-	-	z_5	6	-	t_3'	-	-	t_5'	-
22	t_3'	-	-	-	-	t_5	7	-	x_3'	-	-	x_5'	-
23	x_3'	y_3'	$\mathcal{H}_1$	-	y_5'	x_5	8	-	$\mathbf{y_6}'$	-	-	-	$\mathbf{y_5'}$
24	y_4	y_4'	-	$\mathcal{H}_2$	y_6'	-	9	-	$\mathbf{z_6}'$	-	-	-	$\mathbf{z_5}'$
25	z_4	z_4'	-	-	z_6'	-	10	-	$\mathbf{t_6}'$	-	-	-	$\mathbf{t_5}'$
26	t_4	t_4'	-	-	t_6'	-	11	-	$\mathbf{x_4}'$	-	-	-	$\mathbf{x_5}'$
27	y_3	x_4'	-	$\mathcal{H}_1$	x_6'	y_5'	-	-	-	-	-	-	-

Table 6.5: Device utilization and maximum clock frequency on Xilinx Zynq-7020 FPGA.

Component	**Single-core @138.7 MHz**	**Multi-core @129.2 MHz**	**Available**
LUTs	8,764 (16%)	35,015 (66%)	53,200
Registers	6,852 (6%)	27,300 (26%)	106,400
DSP48E1	49 (22%)	196 (89%)	220
Block RAM	0 (0%)	0 (0%)	140
Occupied slices	2,657 (20%)	10,554 (79%)	$13,300$

Instantiating multiple single-cores is a common concept and was similarly applied by Sasdrich and Güneysu [37] for Curve25519 and Järvinen et al. [35] for Four$\mathbb{Q}$. Sasdrich and Güneysu used a shared inversion module and Järvinen et al. used a shared control logic component. We also implemented a multi-core architecture with a shared control logic using a single shared key to reduce the area utilization. However, the LUT logic was only reduced by approximately 10% which is a rather small improvement compared to its limitations. In fact, this shared control logic architecture requires all scalar multiplications to be started in parallel as there is only one control logic for all cores.

6.5 Results and Analysis

We synthesized our single-core and multi-core architectures with Xilinx Vivado 2017.2 on a Xilinx Zynq-7020 FPGA (XC7Z020CLG484-3). All our results are obtained after place-and-route. Table 6.5 presents the area utilization including the maximum clock frequency for the single-core and multi-core architecture. Our single-core architecture requires 20% of the available slices and 22% of the available DSP blocks. Through according design methods and proper constraining we achieve a maximum clock frequency of 138.7 MHz, which corresponds to a clock period of 7.21 ns. Two interleaved scalar multiplications require 11,330 cycles, and thus a session-key can be computed with a latency of 82 μs. The interleaving of two scalar multiplications can then be either used to effectively double the throughput to 24,482 scalar multiplications per second or provide resistance against fault attacks. For the single-core architecture latency is primarily of interest and thus we assume that only a single input point and secret scalar is available at a time

Table 6.6: Comparison of single- and multi-core architectures of variable-base scalar multiplications featuring a 128-bit security level on a Zynq-7020.

Reference	Curve	Cores	Resources			Latency	TP
			Slices	DSP	BRAM	(μs)	(op/s)
[37]	Curve25519	1	1,029	20	2	397	2,519
[27]	Curve25519	1	6,161	175	0	125	8,027
[35]	Four$\mathbb{Q}$ (Mont.)	1	565	16	7	310	3,222
[35]	Four$\mathbb{Q}$ (End.)	1	1,691	27	10	157	6,389
This work	**Kummer**	**1**	**2,657**	**49**	**0**	**82**	**12,224**
[37]	Curve25519	11	11,277	220	22	397	32,304
[35]	Four$\mathbb{Q}$ (End.)	11	5,697	187	110	170	64,730
This work	**Kummer**	**4**	**10,554**	**196**	**0**	**88**	**91,226**

i.e. the interleaving of two scalar multiplications is used as an additional fault countermeasure. For our multi-core design we instantiate the maximum amount of 4 single-cores on the Zynq-7020 FPGA and use the interleaved scalar multiplication for doubling the throughput. Compared to our single-core design, we see a decrease in the maximum clock frequency; using Vivado tools, we can place-and-route our design with a clock frequency of 129.2 MHz which corresponds to a clock period of 7.74 ns. The reduction of the maximum clock frequency is related to the increased DSP block utilization that are distributed across the entire FPGA. For the multi-core architecture with independently operating single-cores we report a throughput of 91,226 scalar multiplications per second.

Table 6.6 provides a comparison of our results with state-of-the-art scalar multiplication implementations on the same Zynq-7020 FPGA device all featuring a 128-bit security level. We compare our genus-2 results to various genus-1 implementations: the X25519 implementation by Sasdrich and Güneysu [37] as well as our implementation from Chapter 4 Table 4.3 (see [27]), and the Four$\mathbb{Q}$ implementation by Järvinen et al. [35]. Comparing the latency of the single-core designs, our proposed implementation is 1.91-times faster than Four$\mathbb{Q}$ using endomorphisms and 3.78-times faster than Four$\mathbb{Q}$ using the Montgomery ladder. The improvement in latency is related to the increased area utilization i.e. our design demands 1.57-times and 4.70-times more slices than Four$\mathbb{Q}$ using endomorphisms and the Montgomery ladder, respectively. Yet, our implementation performs better than the fastest implementation so far (Four$\mathbb{Q}$ with End.) in both the LUT-latency product (217,787 against 265,487) as well as the DSP-latency product (4,018 against 4,239). Compared to the X25519 implementations, the genus-2 implementation is 4.84-times and 1.52-faster than [37] and [27], respectively. In terms of area, the proposed single-core implementation required 2.32-times fewer slices than our implementation from Chapter 3, but 2.58-times more slices than [37].

Our multi-core architecture with independently operating single-cores offers a throughput that is 1.41-times higher than Four$\mathbb{Q}$ and 2.82-times higher than the X25519 implementation. In terms of latency, we also report the fastest scalar multiplication, i.e. our architecture is 1.93-times faster than Four$\mathbb{Q}$ and 4.51-times faster than X25519. Note that all reported multi-core designs use the maximum number of cores that can be successfully placed on the target device. However, only our multi-core design features fully independent single-cores, i.e. neither the inversion unit, such as the X25519 implementation [37], nor the scalar multiplication unit, such as Four$\mathbb{Q}$ implementation [35], are shared. Also note that we make use of distributed RAM implemented by LUT logic for memory, which leaves a notable amount of BRAM available for other applications. We emphasize that X25519 and Four$\mathbb{Q}$ could also benefit from interleaved scalar multiplication. However, this was not included in the corresponding implementations and thus no results can be compared.

6.6 Conclusions

We presented the first hardware implementation results for a key exchange on the Kummer surface of Gaudry and Schost's genus-2 curve. Although a Kummer surface based key exchange has an increased number of field operations per ladder step when compared to elliptic curves, our presented architectures perform a scalar multiplication with lower latency and higher throughput than any other reported prime-field elliptic curve key exchange featuring a 128-bit security level on a Zynq-7020 FPGA. These results set new records for latency and throughput among state-of-the-art 128-bit secure key exchange implementations known so far, such as Curve25519 [27, 37] and Four$\mathbb{Q}$ [35].

Chapter 7

Isogeny-Based Cryptography

Elliptic and hyperelliptic curve cryptography both rely on the ECDLP, which can be solved by a quantum computer in polynomial time using Shor's algorithm [21]. To derive quantum-secure cryptosystems, other mathematical problems must be applied that are hard to solve even for a quantum computer. Beside popular proposals that rely on hash-based, lattice-based, code-based, and multivariate cryptography, *isogeny-based cryptography* has received considerable attention due its small key sizes. Public-key cryptosystems based on isogenies between elliptic curves have been proposed already in the early 2000s by Teske [63] and Rostovtsev et al. [64]. Stolbunov [65] published the first key agreement protocol using isogenies between ordinary elliptic curves, however, as Childs et al. [66] showed a quantum algorithm exists that can solve the isogeny problem on ordinary curves in sub-exponential time. The first widely considered key exchange is the supersingular isogeny Diffie-Hellman (SIDH) key exchange which was published by Jao and De Feo [5] in 2011 that focuses on isogenies between supersingular curves. This chapter introduces the preliminaries of isogenies followed by a description of SIDH.

Outline. Section 7.1 introduces basic terms that are required for understanding isogeny-based cryptography. Section 7.2 describes the SIDH protocol and Section 7.3 discusses the computational complexity of the applied operations.

7.1 Preliminaries

The quantum-secure SIDH key exchange protocol uses elliptic curve arithmetic, i.e. elliptic curves as mathematical structures and its associated point arithmetic, which is well understood in the ECC domain. However, in order to describe SIDH, further preliminary definitions need to be introduced. Therefore, we provide the reader with a brief description of isogenies, supersingular curves, and ℓ-torsion subgroups. A more detailed description can be found in [67, 41, 68].

7.1.1 Isogenies

Suppose E_1 and E_2 are two elliptic curves with the same cardinality, i.e. $\#E_1 = \#E_2$, and with identity elements $\mathcal{O}_1$ and $\mathcal{O}_2$, respectively. From Chapter 3, we know that for elliptic curve cryptography the identity element is represented by the point-at-infinity ∞. Then an isogeny is a surjective mapping $\phi : E_1 \to E_2$ with $\phi(\mathcal{O}_1) = \mathcal{O}_2$. This mapping is also a group homomorphism, i.e. $\forall P, Q \in E_1 : \phi(P \oplus Q) = \phi(P) \oplus \phi(Q)$. Two elliptic curves are called isogenous if there exists an isogeny between them. The kernel of an isogeny is defined as the set of points on the domain curve that map to the identity element: $\ker(\phi) = \{P \in E_1 \mid \phi(P) \to \mathcal{O}_2\}$. There is a one to one correspondence between isogenies and their kernels, and an isogeny can be computed from its kernel. Using the kernel of an isogeny to store it as a data structure is common in SIDH. As described in [41], if E_1 is an elliptic curve, then for any subgroup $H \subseteq E_1$ there exists a unique (up to isomorphism) elliptic curve E_2 with an associated isogeny $\phi : E_1 \to E_2$ with $\ker(\phi) = H$. This isogeny is a natural map: its image is isomorphic to the quotient of the kernel in the domain, i.e. $E_2 \cong E_1/\ker(\phi)$. Parts of the protocol deal with the computation of an isogeny of a certain degree. For the purpose of this work, the degree of an isogeny is the cardinality of its kernel.

7.1.2 Supersingular Curves

Elliptic curves can be either ordinary or supersingular. An elliptic curve $E(\mathbb{F}_q)$ with $q = p^a$, where p is a prime and $a \in \mathbb{Z}$, is called supersingular if $\#E(\mathbb{F}_q) \equiv 1 \mod p$. Supersingular curves were proven to reduce the computational complexity of the elliptic curve discrete logarithm problem [69], which restricts their application in ECC. However, Childs et al. [66] showed that solving the isogeny problem for ordinary elliptic curves, i.e. finding an isogeny between two known ordinary curves, can be done in quantum-polynomial time. This fact implies that cryptographic protocols based on the ordinary isogeny problem are insecure in the post-quantum world. The opposite is considered to be true regarding the supersingular case [67].

7.1.3 ℓ-Torsion Subgroups

Let $E(\mathbb{F}_q)$ be an elliptic curve defined over a finite field of prime characteristic p. For any integer ℓ the ℓ-torsion subgroup of E is defined as $E[\ell] := \{P \in E \mid [\ell]P = \mathcal{O}\}$ [71]. The ℓ-torsion subgroup of an elliptic curve also has a special structure: $E[\ell] \cong \mathbb{Z}/\ell\mathbb{Z} \times \mathbb{Z}/\ell\mathbb{Z}$. In other words, $E[\ell]$ can be generated by two different points $P, Q \in E$ of order ℓ, i.e.

$E[\ell] = \langle P, Q\rangle := \{[m]P \oplus [n]Q \mid m, n \in \mathbb{Z}\}$. SIDH takes advantage of this structure as will be described in Section 7.2.

7.2 The Supersingular Isogeny DH Key Exchange

Jao and De Feo [5] proposed SIDH as a variant of the Diffie-Hellman key exchange based on the isogeny-graph problem. Similarly to standard Diffie-Hellman, SIDH has a number of public parameters, as described in Section 7.2.1, and is separated into two phases: the key pair and shared secret key computation as presented in Section 7.2.2 and Section 7.2.3, respectively. We shortly describe algorithms for the large degree isogeny computation. This operation is analogous to the scalar multiplication in traditional ECC, and is computed iteratively as detailed in Section 7.2.4.

7.2.1 Public Parameters

Before keys can be exchanged, SIDH requires to fix the base field, the supersingular elliptic curve and some points on this curve.

Base Field

A finite field $\mathbb{F}_q := \mathbb{F}_{p^2}$ is fixed where p is some large prime with the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$. The values ℓ_A and ℓ_B are small primes, and $e_A, e_B, f \in \mathbb{N}$, with f being a cofactor chosen in such a way that p is prime. Alice will compute isogenies of degree $\ell_A^{e_A}$ and Bob will compute isogenies of degree $\ell_B^{e_B}$. Note that it is recommended to chose $\ell_A^{e_A} \approx \ell_B^{e_B}$ to achieve a similar security level and computational complexity for both parties.

Elliptic Curve and Bases

Alice and Bob define a supersingular elliptic curve $E_0(\mathbb{F}_{p^2})$. Next, four points are chosen $P_A, Q_A, P_B, Q_B \in E_0$ fixing the bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ generating the $\ell_A^{e_A}$-, and $\ell_B^{e_B}$-torsion subgroups, respectively: $E_0[\ell_A^{e_A}] = \langle P_A, Q_A\rangle$ and $E_0[\ell_B^{e_B}] = \langle P_B, Q_B\rangle$.

7.2.2 Key Generation

Alice chooses two secret random integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, both not divisible by ℓ_A and computes $R_A := [m_A]P_A \oplus [n_A]Q_A$. It holds that $R_A \in \langle P_A, Q_A\rangle = E_0[\ell_A^{e_A}]$ and thus $\#\langle R_A\rangle = \ell_A^{e_A}$. Alice can then compute the isogeny ϕ_A with $\ker(\phi_A) = \langle R_A\rangle$ and thus $\deg(\phi_A) = \ell_A^{e_A}$ taking E_0 to a new elliptic curve E_A. The isogeny

ϕ_A is the quotient map, so the curve E_A is isomorphic to $E_0/\langle R_A\rangle$. Finally, Alice evaluates the points P_B and Q_B using the isogeny ϕ_A, and saves the values $\phi_A(P_B)$ and $\phi_A(Q_B)$. Bob proceeds *mutatis mutandis.* The triple $(E_A, \phi_A(P_B), \phi_A(Q_B))$ is Alice's public key and the pair (m_A, n_A) is her private key. Furthermore, let $(E_B, \phi_B(P_A), \phi_B(Q_A))$ and (m_B, n_B) be the similarly computed key pair belonging to Bob.

7.2.3 Shared Secret Computation

Alice now has access to Bob's public key $(E_B, \phi_B(P_A), \phi_B(Q_A))$. The goal is to reach some new elliptic curve E_{BA} by computing a new isogeny $\phi'_A : E_B \to E_{BA}$. For this purpose, Alice uses her secret integers (m_A, n_A) and computes the point $S_A \coloneqq [m_A]\phi_B(P_A) \oplus [n_A]\phi_B(Q_A)$. As in the previous phase of the protocol, an isogeny ϕ'_A with $\ker(\phi'_A) = \langle S_A\rangle$ and thus $\deg(\phi'_A) = \ell_A^{e_A}$ can be efficiently computed taking E_B to the final elliptic curve E_{BA}. Bob proceeds *mutatis mutandis* and computes the isogeny ϕ'_B and the elliptic curve E_{AB}. It holds that $E_{BA} \cong E_{AB}$, which implies that their j-invariants $j(E_{BA}) = j(E_{AB})$. Alice and Bob can thus use this common value as a shared secret key. For further details regarding the j-invariants of elliptic curves, we refer the reader to [41].

7.2.4 Large Degree Isogeny Computation

Given an elliptic curve $E(\mathbb{F}_q)$ and a subgroup $H \subseteq E$ with $H \coloneqq \langle R\rangle, R \in E, \mathrm{ord}(R) = \ell^e$, where ℓ is a small prime, one can compute an isogeny ϕ with $\ker(\phi) = H = \langle R\rangle$ and $\deg(\phi) = \#H = \ell^e$. For example, Alice was required to compute ϕ_A with $\ker(\phi_A) = \langle R_A\rangle = \langle [m_A]P_A \oplus [n_A]Q_A\rangle$. To compute this isogeny the problem is divided into smaller operations comparable to decomposing the ECC scalar multiplication into single point additions. The isogeny ϕ can be written as a composition of e isogenies ϕ_i of degree ℓ. The isogeny ϕ is obtained by taking the curve E to a curve isomorphic to the quotient of $\langle R\rangle$ in E, i.e. $\phi : E \to E/\langle R\rangle$. First set $R_0 \coloneqq R$ and $E_0 \coloneqq E$. Then for $0 \le i < e$, the simplest algorithm for the large-degree isogeny computation is the *multiplication oriented* approach and is given by:

$$E_{i+1} = E_i/\langle[\ell^{e-i-1}]R_i\rangle, \quad \phi_i : E_i \to E_{i+1}, \quad R_{i+1} = \phi_i(R_i),$$

with $E_e \cong E/\langle R\rangle$ and $\phi_{e-1} \circ \phi_{e-2} \circ \cdots \circ \phi_0 = \phi$. This means that in each iteration, the current point is multiplied with ℓ until $[\ell^{e-i-1}]R_i$ is determined. Then we can compute the kernel of the isogeny $\ker(\phi_i)$ and subsequently use Vélu's formulas to obtain the isogeny ϕ_i. Next, we push the point R_i through the isogeny ϕ_i to obtain R_{i+1} and repeat the process.

To draw a clearer picture of the structure of the large degree isogeny computation, assume that $\ell = 2$ and $e = 5$. Figure 7.1 illustrates the computational

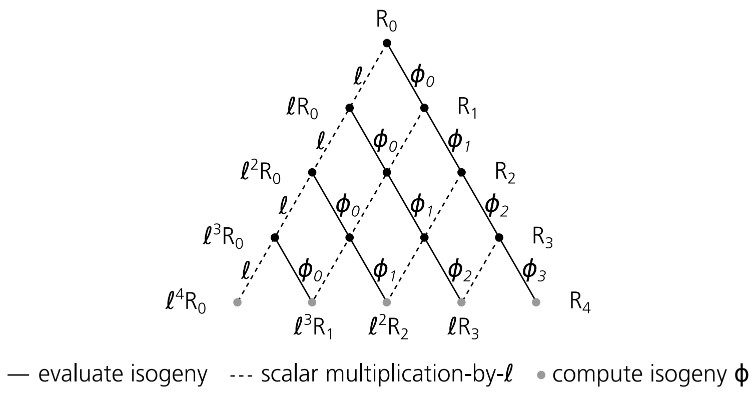


Figure 7.1: Computational structure of the large degree isogeny computation.

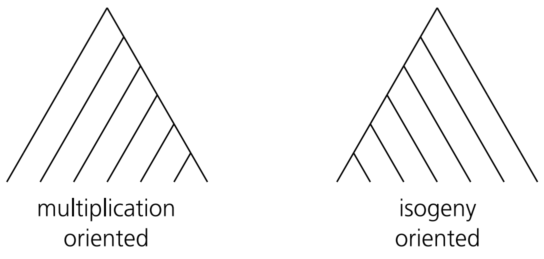


Figure 7.2: Two well-formed strategies for $\ell^e = 2^6$. A strategy is said to be well-formed if it has no useless edges.

structure for this example. It can be easily seen that computing $\langle R_0 \rangle$ is computational complex as the set contains 32 elements. Instead the isogeny ϕ_0 is obtained by computing $\langle [\ell^{e-1}]R_0 \rangle = \langle [2^4]R_0 \rangle$ where with each successive scalar multiplication the order is divided by $\ell = 2$. With this reduced kernel size, the isogeny ϕ_0 can be evaluated using Vélu's formulas [70]. In the next step, we compute $R_1 := \phi(R_0)$, which also divides the order by ℓ.[1] This ultimately leads to a tree structure of the large degree isogeny computation where the objective is to reach all vertices on the bottom line. We finished the large degree isogeny computation, after we computed the isogeny ϕ_4 at edge R_4.

Strategies

Aside from the *multiplication oriented* algorithm, Jao and De Feo [71] also introduced and formally defined the *isogeny oriented* algorithm. In short, instead of relying on point multiplications as the main operation, the *isogeny oriented* approach computes mainly ℓ-isogeny evaluations. Two different strategies, including the isogeny and multiplication oriented, are illustrated in Figure 7.2. As the au-

[1]This follows from $l^{e-1}R_1 = \phi_0(\mathcal{O}_{E_0}) = \mathcal{O}_{E_1}$.

thors of [71] show, both of these approaches are non-optimal, i.e. they carry out more operations than necessary. Instead, they define the concept of an *optimal strategy* as the combination of the two approaches which results in the fewest number of base operations required. Optimal strategies can be computed in advance and stored as constants, as described by [72]. This technique has been used in a number of SIDH implementations, including ours.

7.3 Complexity Considerations

To estimate the run-time of SIDH on various platforms and compare the scheme to other post-quantum algorithms, a complexity analysis is helpful. This complexity analysis breaks down SIDH in single computations and reports the number of field operations in the specified field $\mathbb{F}_{p^2}$. We recall, that the full protocol can be broken down into two rounds, i.e. key generation and shared key computation, where each round follows a similar procedure:

1. Compute $R = [m]P \oplus [n]Q$ for points P, Q.
2. Compute the isogeny $\phi : E \to E/\langle R \rangle$ for the supersingular curve E.
3. Compute the images $\phi(P)$ and $\phi(Q)$ for the basis of the opposite party.

Alice and Bob compute the double point multiplication twice, i.e. during the key generation and shared secret key computation phase. The double point multiplication can be computed by a three-point ladder that computes $P \oplus [n]Q$ in $\mathbb{F}_{p^2}$ without diminishing the security of the protocol, as proposed by Jao and De Feo [71]. The ladder requires $9t\mathrm{M} + 6t\mathrm{S} + (14t + 3)(\mathrm{A/Z})$ operation in $\mathbb{F}_{p^2}$, where t stands for the bit-length of n_A or n_B, M stands for modular multiplication, S for modular squaring, and A/Z for modular addition/subtraction. Note that Alice chooses $n_A \in \mathbb{Z}/\ell_A^a\mathbb{Z}$ and Bob chooses $n_B \in \mathbb{Z}/\ell_B^b\mathbb{Z}$. Therefore, n_A and n_B are differently sized resulting in a different run-time. To enable a better understanding, Table 7.1 (which is a simplified version of [3]) exemplary denotes the required operations for the entire SIDH protocol for $p_{751} = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1 = 2^{372}3^{239} - 1$. As it can be observed 1502 three-point ladder steps are computed, since the bit length of $|n_A| + |n_A| = 751$.

The large degree isogeny computation is the most time demanding operation in SIDH. The main operations in the large degree isogeny computation are: scalar multiplication-by-ℓ, isogeny computation, and isogeny evaluation. As described in the previous Section 7.2.4, different strategies exist, which have an impact on the required number of operations. We begin by stating the cost for the isogeny computation because it is independent of the strategy. The isogeny ϕ is decomposed in smaller isogenies $\phi_{e-1} \circ \phi_{e-2} \circ \cdots \circ \phi_0 = \phi$ and thus e isogeny computations of degree ℓ are required. Hence, 478 3-isogenies are computed for $e_B = 239$. We note that Alice typically computes 4-isogenies instead of 2-isogenies, because it reduces the computational complexity as shown by Jao and De Feo [71]. This means that

Table 7.1: Field operations for SIDH in $\mathbb{F}_{p^2}$ using prime p_{751}, derived but simplified from [3].

Routine	**Operations in $\mathbb{F}_{p^2}$**			**#Operations**
	M	**S**	**A/Z**	
Three-point ladder step	9	6	14	1502
Mont. quadruple	8	4	11	1276
Mont. triple	8	5	15	1622
Compute 4-isogeny	0	5	7	372
Evaluate 4-isogeny	9	1	6	383
Compute 3-isogeny	3	3	8	478
Evaluate 3-isogeny	6	2	2	408
Inversion	196	757	2	4

$372/2 = 186$ isogenies of degree 4 are computed, whose composition has degree $4^{186} = 2^{2\cdot 186} = 2^{372}$. The multiplications-by-ℓ are performed by Montgomery ladder denoted as Montgomery quadruple and triple for $\ell = 4$ and $\ell = 2$, respectively. The number of multiplications-by-ℓ and isogeny evaluations depends on the chosen strategy of the large degree isogeny computation. Computing the images $\phi(P)$ and $\phi(Q)$ is combined with the large degree isogeny computation by evaluating the P_i, Q_i together with the generator R_i. By accumulating all field operations, we observe that SIDH requires approximately 44,064M + 28,245S + 62,185(A/Z) in $\mathbb{F}_{p^2}$ where $p_{751} = 2^{372}3^{239} - 1$.

Chapter 8

SIDH Key Exchange on Embedded Devices

In this chapter, we discuss the practicability and implementation security of SIDH for embedded devices. Therefore, we discuss aspects of related hardware implementations and present two speed-optimized software implementations for a 32-bit ARM Cortex M4 and a 16-bit TI MSP430X. Parts of this chapter are planned for publication [29].

Outline. In Section 8.1 we provide the reader with an introduction. Section 8.2 summarizes related work on existing SIDH hardware accelerators. In Section 8.3 we present our embedded implementation for the Cortex-M4 and the MSP430X with special emphasis on the prime field operations. We summarize our performance results in Section 8.4 and discuss randomized projective coordinates and public key validation in Section 8.5. Finally, we conclude in Section 8.6.

8.1 Introduction

Costello et al. [72] published the first constant-time implementation on Intel Sandy Bridge and Haswell processors using projective coordinates. In terms of speed, their results were recently surpassed by Hernández et al. [73]. In 2016, Koziel et al. [74] presented a highly-optimized implementation in affine coordinates on a comparably less powerful 32-bit Cortex-A8 and Cortex-A15 architecture using the NEON SIMD architecture extension. However, until now it remains unclear how SIDH performs on microcontrollers which possess less computational power and lack dedicated SIMD accelerators. We show that SIDH fails to live up to those expectations due to its inferior performance on relevant controllers of such embedded devices rendering it unsuitable for most real-life applications. Our claim is based on the performance of two speed-optimized implementations for an ARM

Cortex-M4 32-bit microcontroller and a TI MSP430X 16-bit microcontroller both utilizing a 751-bit wide extension field targeting at least a 128-bit quantum and 192-bit classical security level. We base our implementation on Microsoft's published SIDH library, but thoroughly optimize the prime field operations for the corresponding architectures in assembly. Even though our results outperform the generic C-implementation by an order of magnitude, an ephemeral key exchange still requires more than 18 seconds on the ARM Cortex-M4 at 120 MHz and more than 11 minutes on the TI MSP430X architecture at 16 MHz, which is clearly too long. We note that utilizing a smaller extension field could lead to a serious performance improvement enforcing the need for a further evaluation. As ephemeral keys are impractical for those microcontrollers due to long computation time and hence increased energy consumption, static keys are likely to be used. However, the application of static keys can make SIDH vulnerable to vertical unprofiled side-channel analysis. Therefore, we show that randomized projective coordinates, as a countermeasure to thwart DPA, can be implemented for only 3% computational overhead and perform a leakage detection test to demonstrate the effectiveness as part of a case study.

8.2 Related Hardware Implementations

As described in Section 7.3, SIDH features a high computational complexity which makes a dedicated hardware accelerator an appealing choice. Since several speed-optimized works have already been published, we refrain from contributing a further hardware implementation and instead summarize the results of other implementations for comparability. In 2018, Koziel et al. [75] presented a scalable high-performance implementation of SIDH as major extension of their previous works [3, 76, 77]. On a Virtex-7 FPGA, they report area and performance results targeting a 83, 124, 168, and 252-bit quantum security level.

8.2.1 Architecture

Figure 8.1 illustrates the proposed hardware architecture of [75], which can be broken down into the following components:

- Modular multiplier, adder and subtractor in $\mathbb{F}_p$.
- A controller including read-only memory (ROM) for storing instructions.
- RAM (256 entries with the size of an element in $\mathbb{F}_p$) for curve constants and intermediate values.

The hardware implementation can be seen as an application specific instruction set processor (ASIP) due to its CPU similar design. In fact, the authors use their own assembly language including the compilation to 26-bit wide instructions, which are processed by the controller. Those controls are stored inside the ROM.

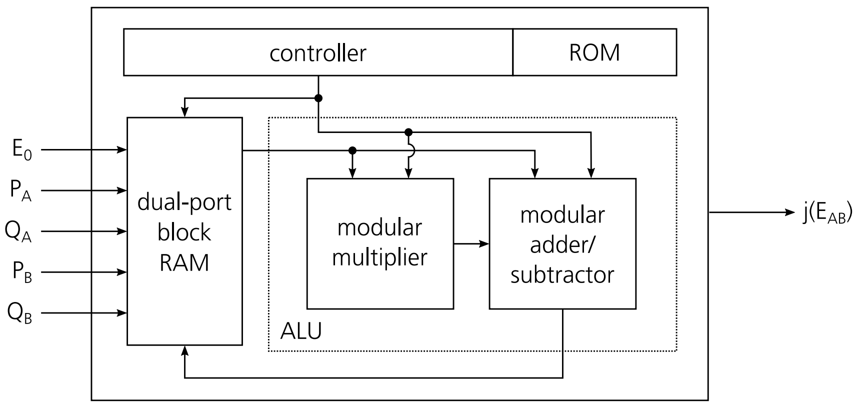


Figure 8.1: Top-level view of hardware architecture [3].

The modules that compute addition, subtraction, and multiplication in $\mathbb{F}_p$ are the performance critical components of the design. Koziel et al. [75] instantiated a highly-optimized modular multiplier, adder and subtractor unit that is centered around a dual-port RAM for storing 256 values in $\mathbb{F}_p$. Being capable of storing 256 values in $\mathbb{F}_p$ allows for storing 128 values in $\mathbb{F}_{p^2}$.

Modular Adder/Subtractor

The finite field addition computes $C = A + B$ where $A, B, C \in \mathbb{F}_p$. If the result is greater than C, then a reduction is computed by $C = C - p$. Due the large field size, the addition/subtraction itself is split into 256-bit chunks for reducing the longest critical path. The entire module is fully pipelined i.e. a new operation can be scheduled each cycle resulting in a chain of addition modules. The reduction step is always computed and the correct result is selected at the end of the addition chain. Thus, the entire modular addition requires 3 cycles in a 751-bit wide finite field. The modular subtraction is computed in a similar fashion.

Modular Multiplier

Koziel et al. [75] instantiate a Montgomery multiplier [32], which is well suited for large field arithmetic and has hence a long tradition in the RSA [7] hardware accelerators. Montgomery multiplication requires to transform integers to the Montgomery domain. Therefore, the proposed SIDH architecture initially transforms all input operands to the Montgomery domain and uses Montgomery multiplication throughout the protocol. After the respective computations are finished, the result is converted back. The implemented modular multiplier is a so-called interleaved systolic Montgomery multiplier as proposed in [77]. This multiplier computes a single modular multiplication in 99 cycles and can interleave a second modular

Table 8.1: SIDH area utilization and performance results on a Virtex-7 FPGA compared to X25519 and Kummer on a Zynq-7020 FPGA.

Work	**Prime**	**Multipliers**	**Resources**			**Latency**
			Slices	**DSP**	**BRAM**	**(ms)**
SIDH [75]	$2^{372} \cdot 3^{239} - 1$	6	11,277	288	61	36.4
SIDH [75]	$2^{372} \cdot 3^{239} - 1$	8	14,447	384	59	33.7
SIDH [75]	$2^{372} \cdot 3^{239} - 1$	10	16,983	480	56	33.2
SIDH [75]	$2^{372} \cdot 3^{239} - 1$	12	19,892	576	55	31.6
Kummer	$2^{127} - 1$	1	2,657	49	0	0.33
X25519	$2^{255} - 19$	1	6,161	175	0	0.5

multiplication at cycle 68. Koziel et al. [75] instantiated a variable number of multiple Montgomery multipliers as a trade-off between area and performance. The modular multiplication is also used for computing squaring operations in $\mathbb{F}_p$.

8.2.2 Results

Table 8.1 shows the area utilization and performance results for SIDH using the $p_{751} = 2^{372} \cdot 3^{239} - 1$ prime. The core was synthesized using Vivado 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 device. Results were obtained for varying number of modular multipliers in range from 6 to 12. Note that the latency results are reported for an entire key exchange i.e. key generation and shared secret computation for Alice and Bob. When comparing 6 modular multipliers to 12 modular multipliers, the latency has improved by approximately factor-1.15. On the other hand, the slice utilization has nearly doubled. For the prime p_{751}, a latency of of 31.6 ms is achieved, which is two times faster than an Intel Haswell software implementation [72]. When compared to work from Chapter 4 and 6, the X25519 and the Kummer surface based implementation have an improved latency by factor-66 and factor-95, respectively. Moreover, both our designs feature a significantly lower area utilization.

8.3 Embedded Implementations

In this section, we provide the reader with a detailed description of our speed optimized implementation for two embedded platforms. We begin by describing the platform independent design decisions in Section 8.3.1. This is followed by a summary of the features of the two microcontrollers in Section 8.3.2 and a detailed description on the implementation of the prime field arithmetic for the corresponding architectures in Section 8.3.3.

8.3.1 Platform Independent Design Decisions

In the following, we summarize a selection of design decisions that we made for our implementation:

Projective Coordinates

As with traditional ECC, projective coordinates speed up each scalar point multiplication (performed twice for an ephemeral key exchange) as it reduces costly field inversions. Costello et al. [72] showed that a more compact representation is derived when operating on variable curve parameters represented in the projective space. Additionally, we represent curve points in projective coordinates and randomize them during scalar multiplication as a computationally efficient countermeasure to thwart DPA.

Structure of Public Keys

To limit the communication overhead and save resources such as energy, the size of the public key should be small. Compared to the initial proposal by Jao and De Feo [5], we follow Costello et. al [72] where the size of the public key is reduced from 768 bytes to 564 bytes. More precisely, the public key is a triple of the field elements in $\mathbb{F}_{p^2}$, representing the x-coordinates of $\phi_A(P_B), \phi_A(Q_B), \phi_A(Q_B \ominus P_B)$ as an example for Alice. The normalized Montgomery curve parameter A of the public curve is recovered from those three points on the curve, and does not need to be included. Note that in [24] it was shown that the public key can be further reduced to only 330 bytes. However, we discarded this technique because it reduces the speed by more than a factor of 3, which collides with our optimization preference for speed.

Chosen Parameters

The characteristic of the field $\mathbb{F}_{p^2}$ is $p = 2^{372} \cdot 3^{239} - 1$ with $\lceil \log_2(p) \rceil = 751$. This prime precisely provides a 124-bit quantum security level, however, it is usually associated to a 128-bit quantum security level. Other primes are proposed in [74] such as $2^{250} \cdot 3^{159} - 1$ and $2^{493} \cdot 3^{307} - 1$ that provide a 83-bit and 162-bit quantum security level, respectively. We decided to target the 128-bit quantum security level, as it is considered to be reasonable secure for the next few decades, while being small enough for sufficient speed. The bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ are set by the following points: $P_A = [3^{239}](11, \sqrt{11^3 + 11})$, $Q_A = \tau(P_A)$, $P_B =$

$[2^{372}](6, \sqrt{6^3+6})$, $Q_B = \tau(P_B)$, where $\tau : E_0 \to E_0$ and $\tau(x,y) = (-x, iy)$. The base supersingular elliptic curve has the short Weierstrass form:

$$E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x. \tag{8.1}$$

One Scalar as Private Key

Instead of choosing two randomly distributed integers m_A and n_A and computing the secret isogeny whose kernel is $\langle [m_A]P_A \oplus [n_A]Q_A \rangle$, Alice chooses one single integer $m_A \in [1, 2^{371}-1]$ and the isogeny with the kernel $\langle P_A \oplus [2m_A]Q_A \rangle$. Similarly, the kernel of Bob's secret isogeny will by $\langle P_B \oplus [3m_B]P_B \rangle$, where $m_B \in [1, 3^{238}-1]$. This is done in order to facilitate the use of pre-computed strategies for isogeny computations. As Costello et al. [72] point out, this reduces the total number of possibilities for the public key by a factor of 3, for Alice, and a factor of 4, for Bob. However, the authors claim there is currently no reason to believe the security of the system is affected by this implementation choice.

8.3.2 Microcontrollers

For development and testing purposes, we used the MSP430FR5994 launchapd and FRDM-K64F development board, that feature two popular 16-bit and 32-bit microcontroller architectures, respectively:

TI MSP430FR5994

The TI MSP430FR5994 is based on the 16-bit MSP430X architecture running at a maximum clock frequency of 16 MHz with 8 kB of RAM and 256 kB of non-volatile FRAM (Ferromagnetic Random Access Memory). The FRAM can be accessed at a frequency of 8 MHz and can be used for long-term storage, as well as machine code and data storage. When the core is clocked with 16 MHz, additional wait cycles are introduced if FRAM access is required due to the difference in the two operating clock frequencies. This can effect the overall performance and is described in the results section. We used Code Composer Studio for code development and compilation with optimization level set to speed.

Kinetis K64

The Kinetis K64 is based on the 32-bit ARM Cortex-M4 core running at 120 MHz with 1 MB of flash memory and 512 kB of RAM. The compilation was done using the GNU ARM Embedded toolchain with optimization set to $-$O3.

8.3.3 Finite Field Operations

As discussed in Section 7.2.1, SIDH defines elliptic curves over the extension field $\mathbb{F}_{p^2}$. Yet, operations in the extension field $\mathbb{F}_{p^2}$ are composed of operations in the finite field $\mathbb{F}_p$. Since the performance of operations in $\mathbb{F}_p$ has strong impact on the overall performance, it is crucial to optimize them for best speed results. The relevant operations are addition, subtraction, multiplication, and modular reduction. All operations run in constant-time, are written in assembly with fully unrolled loops and no calls to subroutines.

Addition and Subtraction

The modular addition and subtraction correspond to standard 24-limb and 48-limb operations for the 32-bit Cortex-M4 and the 16-bit MSP430X, respectively. The *limb notation* describes how many 16 or 32-bit words we require to store an element. Note that both the operands and the result will be elements in $[0, 2p - 1]$, instead of $[0, p - 1]$. As [63] points out, this circumvents the necessity of a subtraction at the end of the modular operation. After an addition or subtraction has taken place, the result has to be reduced to $[0, 2p - 1]$. Since $a, b < 2p$, it holds that $c := a + b < 4p$, i.e. the bitlength of $a + b$ is higher by at most the carry bit. If $c > 2p$, then $c - 2p \in [0, 2p - 1]$ will be the correct result. In order to avoid conditional branching, instead of comparing c to $2p$, the use of the following well-known strategy is employed:

1. Set $c \leftarrow c - 2p$, and remember the borrow bit b.
2. Compute the bitmask $m := (b \,\&\, 2p)$, and set $c \leftarrow c + m$.

Modular subtraction is computed in a similar fashion.

Multiplication

We decided to use Karatsuba multiplication [78] because it has a time complexity of only $\mathcal{O}(n^{\log_2 3})$; for comparison, the standard schoolbook multiplication has a time complexity of $\mathcal{O}(n^2)$. More precisely, we implemented a 1-level additive Karatsuba multiplication with Comba optimizations [79]. The purpose of the latter is to decrease expensive memory accesses and storage requirements for intermediate results. With these optimizations, the memory space dedicated to the result is only accessed when the final value for a specific limb has been computed.

In Karatsuba multiplication, two n-digit operands x, y represented in some base R are split into two parts each: the top (most significant) halves x_H, y_H, and the bottom (least significant) halves x_L, y_L. Define:

$$\begin{aligned} H &:= x_H \cdot y_H \\ L &:= x_L \cdot y_L \\ M &:= (x_H + x_L) \cdot (y_H + y_L) - L - H\,. \end{aligned}$$

Then the following holds:

$$x \cdot y = H \cdot R^n + M \cdot R^{n/2} + L\,. \tag{8.2}$$

In our case, the operands x, y are 768 bits (96 bytes) long, in either 48-limb representation on the MSP430X, i.e. $n = 48, R = 2^{16}$, or 24-limb representation on the Cortex-M4, i.e. $n = 24, R = 2^{32}$. The result is stored in z, which is a $768 \cdot 2 = 1536$ bits (192 bytes) memory location. The most significant words are stored first. In order to store intermediate results, 96 bytes of stack space are allocated at the beginning of the routine. After determining all partial multiplications, M can now be computed by subtracting H and L from the result of the first multiplication. The first product is stored in the first 96 bytes of the allocated stack space, so when subtracting H and L, save the results in the remaining 96 bytes. Afterwards, M can be added to the middle part of z as per Equation (8.2). This spans the bytes 49-144 of z. Lastly, add the overflow resulting from the last digit addition, as well as any further overflows this operation might produce to the remaining bytes of z, in sequence.

We note that optimizing the multiplication by exploring further algorithms could potentially result in a performance improvement. For example, one could implement multi-level Karatsuba as well as exotic, microcontroller-optimized multiplication algorithms [80].

Reduction

The modular reduction is an adaptation of the well-known Montgomery reduction [32]: let $\mathbb{F}_p$ be the base field with $p = 2^{372} \cdot 3^{239} - 1$, $\lceil \log_2(p) \rceil = 751$, and define $R := 2^{768}$ and $p' = -p^{-1} \mod R$. For any input $a < pR$, compute the Montgomery residue $c = aR^{-1} \mod p$:

$$c := (a + (ap' \mod R) \cdot p)/R\,. \tag{8.3}$$

This operation is generally computed iteratively: first define r as the bitsize of an integer, and set s such that $R = 2^{r \cdot s}$. In this case, $R = 2^{768}$, so for the Cortex-M4, $r = 32, s = 24$, and for the MSP430X, $r = 16, s = 48$. Set $c \leftarrow a$, then repeat s times: $c \leftarrow (c + (c \cdot p'' \mod 2^r) \cdot p)/2^r$, where $p'' = -p^{-1} \mod 2^r$.

As Costello et al. [72] showed, Equation (8.3) can be converted for the chosen prime $p = 2^{372} \cdot 3^{239} - 1$ to:

$$c = a/2^{768} + ((ap' \mod 2^{768}) \cdot 3^{239})/2^{396}\,,$$

Table 8.2: Cycle count (CC) for the prime field operations of the generic and assembly implementation on both architectures.

Operation	Cortex-M4		MSP430X	
	C (CC)	ASM (CC)	C (CC)	ASM (CC)
Mod. Add.	10,779	559	18,500	1,192
Mod. Sub.	7,109	419	12,568	831
Mod. Mul.	244,209	4,319	945,252	32,517
Mod. Red.	167,619	3,254	586,596	20,094

which decreases the number of required multiplications for a modular reduction. Furthermore, they show that in the iterative process, it holds that $p'' = 1$, which allows the transformation: $c \leftarrow (c+(c \mod 2^r)\cdot(p+1))/2^r$. This is advantageous, because in this case, $p + 1$ has a number of its least-significant limbs equal to 0 (11 limbs in 32-bit representation and 23 limbs in 16-bit representation), and they can thus be excluded from the multiplication.

8.3.4 Results for the Assembly Optimized Field Operations

In Table 8.2, we present the number of clock cycles for each field operation for future reference. We implemented the described algorithms in assembly and compare the performance with the generic C implementation by Costello et al. [72], which we ported to our microcontrollers without further modification. It can be noted that our optimized operations require between 15 and 56 times fewer cycles than their generic counterparts. The speed-up of the assembly implementations is comparable for both architectures, while the difference in performance is linked to the architecture dependent word size. The improvement factor is higher for the Cortex-M4, which is likely a result of its lower cycle requirement when accessing consecutive memory locations. Both the generic and the optimized operations run in constant-time.

8.4 Results and Analysis of Constant-Time Implementations

In this section, we first report and compare our results for an ephemeral key exchange to other SIDH implementations. This comparison should aid the reader to classify our results and verify their soundness. Subsequently, we compare our implementation to other quantum-secure key exchange algorithms on embedded

Table 8.3: Clock cycle count [$\times 10^6$] for SIDH on different processors supporting a 128-bit quantum security level.

Work	**Platform**	**Word size**	**Key gen.**		**Secr. gen.**	
			Alice	**Bob**	**Alice**	**Bob**
[73]	Intel Skylake	64-bit	27	31	25	29
[73]	Intel Haswell	64-bit	38	43	34	40
[72]	Intel Haswell	64-bit	51	59	47	57
[81]	ARM Cortex-A57	64-bit	103	118	97	113
[74]	Cortex-A15	32-bit	437	474	346	375
This work	Cortex-M4	32-bit	1025	1148	967	1112
This work	MSP430X (8 MHz)	16-bit	4260	4855	4020	4658
	MSP430X (16 MHz)	16-bit	5136	5824	4832	5600

devices in order to evaluate our work in a broader context.

Table 8.3 compares the clock cycle count for the key pair generation and the shared secret key computation on the Cortex-M4 and the MSP430X to other published SIDH implementations. Note that the clock cycle count differs for Alice and Bob because the computational complexity depends on the selected prime $\ell_A^{e_A}, \ell_B^{e_B}$. For the 32-bit Cortex-M4, the code is compiled to a size of 71.53 kB, and key pairs are generated in 1025 and 1149 million clock cycles for Alice and Bob, respectively. Similar numbers are obtained for the shared secret key computation. For the 16-bit MSP430X microcontroller, we obtained a code size of 110.33 kB. The clock cycle count is reported for two different clock frequencies to show the effect of the introduced wait cycles linked to the lower clock frequency of the FRAM. In case of 8 MHz clock frequency, a key pair key is computed in about 4559 million cycles and a shared secret in about 4339 million cycles. The number of clock cycles increases to about 5480 and 5216 million clock cycles for key pair generation and shared secret key computation, respectively, when being clocked with 16 MHz.

Compared to the performance of the Cortex-M4, the MSP430X requires about 4-times more clock cycles which is linked to the reduced word size of 16-bit. A similar relation is observed when we compare the 64-bit Cortex-A57 [81] and the 32-bit Cortex-A15 [74] implementation, indicating the plausibility of our results. Comparing the 32-bit Cortex-A15 implementation to our implementation on the Cortex-M4, the key generation and shared secret computation requires about 2.38-times and 2.79-times less cycles, respectively. Note that the Cortex-A15 core is based on the ARMv7 architecture and is equipped with features such as caches and the NEON SIMD architecture extension. The lack of such accelerator features explains the increase in clock cycles for our Cortex-M4 implementation. Most

Table 8.4: Performance evaluation of different quantum-secure key exchange protocols on mid- and low-end processors.

Protocol	**Platform**	**Freq.**	**Latency (s)**		**Comm. (bytes)**	
		(MHz)	**Alice**	**Bob**	**A→B**	**B→A**
NewHope [82]	Cortex-M0	48	0.03	0.04	1824	2048
NewHope [82]	Cortex-M4	164	0.01	0.01	1824	2048
Frodo [83]	Cortex-A8	1000	0.08	0.08	11296	11288
SIDH [74]	Cortex-A8	1000	1.41	1.53	564	564
	Cortex-M4	120	16.59	18.83	564	564
SIDH (this work)	MSP430X	8	1035.00	1188.00	564	564
	MSP430X	16	623.00	714.00	564	564

works optimized SIDH for 64-bit processors [72, 73, 81] making a comparison with smaller devices, such as the 16-bit MSP430X, unfair. On 64-bit processors, the current speed record for constant-time implementations is set by Jalali et al. [81], which represents an optimized version of the work by Costello et al. [81].

Comparison

In Table 8.4 we compare other quantum-secure key exchange protocols on embedded devices with our implementation. Relevant parameters are performance in terms of required time measured in seconds and communication overhead measured in transmitted bytes. All listed implementations feature a similar security level of around 128-bit. NewHope [82] was implemented on the ARM Cortex-M4 and Cortex-M0 where an ephemeral key exchange is executed in only 0.01 and 0.035 seconds, respectively. Even when comparing our Cortex-M4 implementation to NewHope on the less powerful Cortex-M0 (clocked with only 48 MHz), NewHope is more than 500-times faster with only 4-times higher communication overhead. Frodo [83] is a LWE-based quantum-secure key exchange with promising performance results as well. For smaller processors, there is only one implementation available for the Cortex-A8, however, its communication overhead implies that implementing it on constrained devices might be impractical. The SIDH implementation on the Cortex-A8 by Koziel et al. [74] shows tolerable execution time and indicates the general applicability of SIDH on such processors. However, compared to NewHope [82] or Frodo [83] the tremendous difference in speed becomes apparent. We conclude that SIDH has small key sizes but clearly suffers in speed, which leads to extensive computation time on small microcontrollers.

8.5 Implementation Security

Contrary to other PQC algorithms (e.g. NTRU [84]), SIDH supports perfect forward-secrecy; however, this also requires the use of ephemeral keys. While forward-secrecy is a desirable property, the secure use of static keys is important for embedded devices due to limited computational power and energy budget. It is well known that elliptic curve based cryptosystems can be attacked by invalid point attacks [51], where a maliciously generated point is used to gain access to the secret private key. To thwart this type of attack, the received points must be validated, i.e. the received point must generate a group with sufficiently large cardinality. As it turns out, validation techniques in the context of SIDH are not trivial: they are either computationally efficient and insecure [72, 85], or secure and computationally inefficient [86]. For example, Kirkwood et al. [86] proposed a working validation technique, which requires as much time as an ephemeral key generation. Therefore, we decided to neglect the implementation of point validation techniques. However, with on-going research we expect computational efficient and secure point validation techniques to be found.

While a point validation technique is the first mandatory step towards the secure use of static keys, a software designer should be aware that static keys can facilitate some attacks. As attackers can typically get physical access to embedded devices, we consider side-channel analysis as an additional attack vector. When static keys are used, an attacker can acquire multiple traces using the same key. Therefore, we evaluate randomized projective coordinates in greater detail in Section 8.5.1 as a countermeasure for preventing DPA [87].

8.5.1 Randomized Projective Coordinates to Thwart DPA

The shared secret computation phase poses a natural target for an attacker because he can control data which is directly processed with the secret private key i.e. the input point that is multiplied with the secret integer during elliptic curve scalar multiplication. DPA on this standard elliptic curve scalar multiplication is well understood. As explained in Section 8.3.1, we only use one integer as our secret scalar for the point multiplication. Here, we target Alice's secret integer n_a and assume that Bob is the malicious entity and can modify $\phi_B(P_A)$, $\phi_B(Q_A)$.

$$S_A = \phi_B(P_A) \oplus [n_A]\phi_B(Q_A)\,.$$

The scalar multiplication and the additional point addition is carried out using the three-point ladder as described by Jao and De Feo [5] and shown in Algorithm 12 (ladder_3pt). Compared to the standard Montgomery ladder, i.e. Algorithm 4 as described in Section 3.4 , the three-point ladder computes $\phi_B(P_A) \oplus [n_A]\phi_B(Q_A)$ directly, hence resulting in improved performance. Note that the if-clause is only

Algorithm 12 ladder_3pt: Three-point ladder [5].

```
Input: $(k = \sum_{i=0}^{|k|-1} k_i 2^i) \in (2^{|k|-1}, 2^{|k|}]$, $P, Q \in E$
Output: $R \leftarrow P \oplus [k]Q$
1: $R_1 = \mathcal{O}, R_2 = Q, R_3 = P$
2: for $i$ from $|k|-1$ to $0$ do
3:     if $k_i = 0$ then
4:         $R_1 \leftarrow 2R_1, \quad R_2 \leftarrow R_1 \oplus R_2, \quad R_3 \leftarrow R_1 \oplus R_3$
5:     else
6:         $R_1 \leftarrow R_1 \oplus R_2, \quad R_2 \leftarrow 2R_2, \quad R_3 \leftarrow R_2 \oplus R_3$
7:     end if
8: end for
9: return $R_3$
```

used here for readability purposes; in our and most other implementations it is replaced by constant-time point swap to prevent SPA and timing attacks.

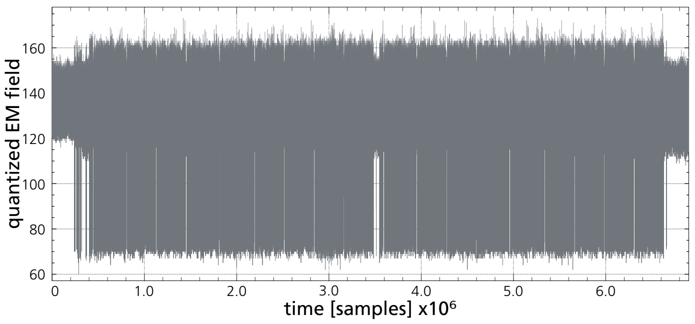
As already discussed in Chapter 4, Coron [50] described *randomized projective coordinates* as an appropriate countermeasure to thwart DPA. This countermeasure is characterized by relatively low computational overhead. Implementing randomized projective coordinates implies a randomly generated λ being multiplied with the input points P, Q in their projective representation during the ladder initialization. Using Montgomery formulas [4], differential point addition for the x-coordinate is given by:

$$X_{P\oplus Q} = Z_{P\ominus Q}[(X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q)]^2 \,,$$

with the two input points $P = \{X_P, Z_P\}$ and $Q = \{X_Q, Z_Q\}$. Due to normalization, the difference point $(Z_{P\ominus Q})$ can be neglected but equals λ for randomized projective coordinates, which translates to one additional multiplication for each point addition. As shown in Algorithm 12, two point additions are performed in each ladder step; thus, enabling randomized projective coordinates results in $744 = 2 \cdot 372$ and $758 = 2 \cdot 379$ additional multiplications in $\mathbb{F}_{p^2}$ for Alice and Bob, respectively. Compared to an unprotected implementation, we require only about 3% more cycles with randomized projective coordinates. This renders randomized projective coordinates a computationally efficient countermeasure.

Case Study: Leakage Assessment on the FRDM-K64F

The Montgomery ladder combined with randomized projective coordinates is considered to be an effective countermeasure to thwart DPA. Even though we expect similar protection for the three-point ladder, a case study is useful for supporting this claim. We acquire EM traces with a Langer RF-B 3-2 near H-field probe (horizontal) placed above the packaged chip. For each implementation, we collect 2500



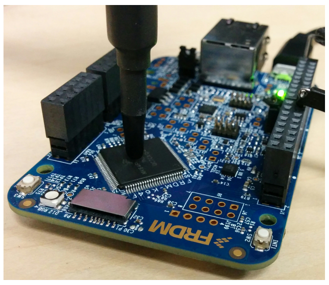


Figure 8.2: Left: Exemplary EM trace for two ladder steps. Right: Langer probe placed above FRDM-K64F.

synchronized traces per measurement at at sampling rate of 5 GS/s using a LeCroy WavePro 725 Zi oscilloscope. We evaluate randomized projective coordinates on the FRDM-K64F (featuring the Cortex-M4) using the non-specific t-test as the leakage detection test [88, 89]. Figure 8.2 shows on the left two ladder steps of the Montgomery ladder, and on the right the probe placed above the FRDM-K64F.

The t-test can be used to detect whether the device's implementation has exploitable leakage. We first test and show that the device leaks secret information with no DPA countermeasure enabled. With the same measurement setup, we then evaluate the leakage with randomized projective coordinates. We apply a *fixed-vs-random* methodology on the input point, i.e. we acquire 2500 traces with a fixed input point and 2500 with a random input point; subsequently, the t-test determines whether the two data sets are significantly different to each other. The input point and the random number λ are sent to the development board via UART while the secret remains fixed. In case of the unprotected implementation, we fix λ to a constant value. Figure 8.3 shows on the left the t-test with no DPA countermeasures and on the right with randomized projective coordinates. With no countermeasures enabled, the device fails the t-test as it exceeds the threshold $\pm C = 4.5$, which clearly indicates leakage. On the contrary, the test results after the introduction of randomized projective coordinates indicate the effectiveness of the countermeasure as expected.

8.6 Conclusions

We presented two implementations of SIDH targeting a 128-bit quantum security level for the 32-bit ARM Cortex-M4 and 16-bit TI MSP430X architectures that perform the shared secret key computation including key pair generation in about 18 seconds and 11 minutes, respectively. Although our results only set a first benchmark, we conclude that even the inferior performance results of the unprotected implementations indicate that SIDH over a 751-bit wide finite field is

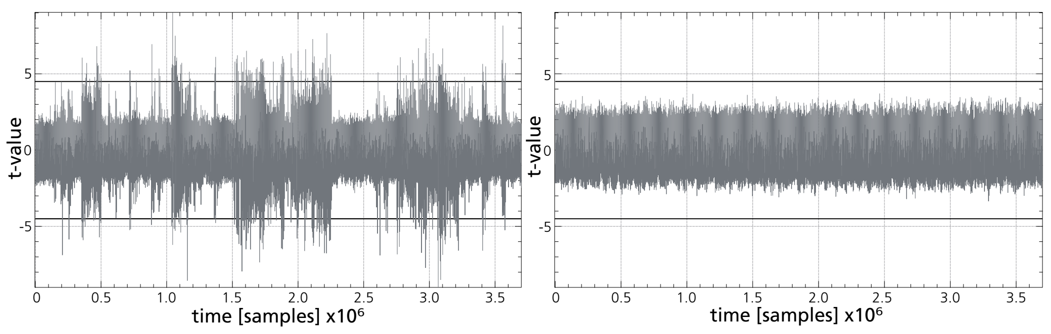


Figure 8.3: Fixed-vs-random leakage detection test on the input point using 5000 traces. Left: no DPA countermeasure. Right: randomized projective coordinates enabled.

impractical for securing resource-constrained devices. It is likely that our implementations can be optimized by a small factor, but it seems to be unrealistic that the performance can be drastically improved. We use randomized projective coordinates to thwart multi-trace DPA as it only reduces the speed by approximately 3%. However, we note that current point validation techniques imply tremendous performance loss emphasizing the need for further research. Moreover, other quantum secure key encapsulation protocols (such as NewHope [82]) seem more suitable for embedded devices. Yet, SIDH may represent a suitable fit for securing the Internet communication where typically more powerful processors are used.

Due to the high computational complexity of SIDH, a dedicated hardware core can greatly accelerate the run-time of SIDH. As various papers showed [3, 76, 77], the latency as well as throughput can be improved by a tremendous factor when compared to the embedded implementation. The reported latency of up to 36.4 ms seems to fit into various scenarios. On the other hand, when compared to state-of-the-art ECC implementations, the area utilization and latency are increased. We positively note that due to the similarity between ECC and SIDH, existing ECC hardware accelerators could be re-used to improve the performance by outsourcing parts of the protocol to the hardware cores.

Chapter 9

Conclusions

In this thesis, an efficient modular multiplier and two optimized implementations of elliptic and hyperelliptic curve cryptography on FPGA were presented. We reported the performance results and area utilization of all implementations and included countermeasures to thwart timing and power analysis attacks. The speed-optimized modular multiplier set the foundation for the X25519 as well as the Kummer surface based key exchange implementation. We further discussed the implementation of the quantum-secure supersingular isogeny Diffie-Hellman key exchange on embedded devices including countermeasures to thwart DPA and verified their effectiveness by measuring the EM radiation.

The modular multiplier is a key component for obtaining high-speed designs since its performance greatly influences the overall computation time. Therefore, we presented a novel hardware design for the multiplication in Mersenne prime fields based on a new optimization strategy of the adder tree and reduction circuits at the bit-level. Our proposed method can be applied for ASIC as well as FPGA designs. Compared to related work, our presented modular multiplier can operate at higher frequencies, leading to improved throughput and latency. We provided a formalization of our proposed strategy for any Mersenne prime and any size of the underlying small-sized (potentially asymmetric) multipliers.

We explored different hardware design strategies for X25519 on two Xilinx Zynq FPGAs targeting low-latency. Thereby, we demonstrated the effectiveness of the developed modular multiplier as it outperformed comparable implementations. The X25519 FPGA implementation, which applies the aforementioned Mersenne prime multiplier, achieved sufficient latency results but suffered in throughput and area.

For applications that have very strict latency and throughput requirements, the hyperelliptic variant i.e. using the Kummer surface of Gaudry and Schost's genus 2 curve, represents an interesting choice. Our presented implementation shows that a key exchange can be performed with lower latency and higher throughput than any other reported implementation. When compared to the X25519 implementation,

Table 9.1: Performance comparison of our implementations using curve based cryptography.

Work	**Platform**	**Cores**	**Frequency (MHz)**	**Latency** (s)	**Throughput (op/s)**
X25519	Zynq-7020	1	84	125×10^{-6}	8000
Kummer	Zynq-7020	1	139	82×10^{-6}	12224
	Zynq-7020	4	129	88×10^{-6}	91226
SIDH	Cortex-M4	1	120	17	0.06
	MSP430	1	16	1035	0.001

the Kummer variant achieves an improved latency that is 1.5-times lower while also featuring a reduced area utilization. Moreover, we showed in the Kummer architecture how two scalar multiplications can be smartly combined for doubling the throughput or providing an inherent countermeasure against fault attacks. This technique might be considered for future curve implementations.

Finally, we presented an implementation of SIDH on popular microcontrollers as they are typically deployed in the IoT. SIDH is a promising candidate because it uses relatively small keys, however, a key exchange requires more than 18 seconds on a 32-bit Cortex-M4 and more than 11 minutes on a 16-bit MSP430 controller, respectively. This is clearly too long for most real-life applications. On a positive note, we also analyzed the implementation security of SIDH and found that appropriate DPA countermeasures can be implemented with little overhead. Moreover, related work showed that the application of dedicated hardware accelerators or more powerful CPUs yield promising performance results. Due to the similarity of SIDH and ECC, existing ECC hardware accelerators could be re-used to improve the performance by outsourcing parts of the protocol to the hardware core. We further emphasize that SIDH is a relatively novel cryptographic algorithm, and thus further improvements in its performance can be expected. Table 9.1 summarizes the performance results of our curve based implementations.

In summary, we analyzed and compared various cryptosystems based on curve based cryptography, i.e. elliptic, hyperelliptic, and isogeny-based cryptography regarding their implementation characteristics. Table 9.2 summarizes their key properties in terms of field type, size, elements, secrets, and underlying mathematical problem. All three feature relatively small key sizes, but differ in their underlying mathematical problem and computational complexity. While elliptic and hyperelliptic curve cryptography feature an efficient arithmetic, isogeny-based cryptography is characterized by a high computational complexity. In addition to that, the underlying field size of SIDH is 2.95-times and 5.91 higher than X25519

Table 9.2: Conceptual comparison between elliptic, hyperelliptic, and isogeny-based cryptography.

	ECC	**HECC**	**Isogeny-based**
Reference	Curve25519 [12]	Kummer [18]	SIDH [5, 71]
Elements	$x(P)$ on E	$\kappa(D)$ on $\mathcal{K}$	Curve E
Field	$\mathbb{F}_p$	$\mathbb{F}_p$	$\mathbb{F}_{p^2}$
Field size	255	127	751
Classical sec.-level	128	128	192
Quantum sec.-level	$\times$	$\times$	128[1]
Secret	scalar k	scalar k	isogeny ϕ
Hard problem	given $P, k[P]$ find k	given $D, [k]D$ find k	given $E, \phi(E)$ find ϕ

[1] Post-quantum level holds at the time of writing but may vary with progress made in cryptoanalysis.

[12] and the Kummer [18] algorithm, respectively. SIDH uses a quadratic extension field, while X25519 and Kummer are defined over a standard prime field. The presented analysis and implementation results can serve as reference for selecting an appropriate cryptosystem with respect to area, performance, and (quantum) security requirements.

Bibliography

[1] Shreesha Srinath and Katherine Compton. Automatic generation of high-performance multipliers for FPGAs with asymmetric multiplier blocks. In *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays, FPGA 2010, Monterey, California, USA, February 21-23, 2010*, pages 51–58, 2010.

[2] Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, and Junko Takahashi. Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 177:1–177:6, 2014.

[3] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Fast hardware architectures for supersingular isogeny Diffie-Hellman key exchange on FPGA. In *Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings*, pages 191–206, 2016.

[4] Peter L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[5] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, pages 19–34, 2011.

[6] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[7] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[8] Dan Boneh. The decision Diffie-Hellman problem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, pages 48–63, 1998.

[9] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[10] Victor S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, pages 417–426, 1985.

[11] Bundesamt für Sicherheit in der Informationstechnik. Cryptographic mechanisms: Recommendations and key lengths (BSI TR-02102-1), 2018.

[12] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *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*, pages 207–228, 2006.

[13] Neal Koblitz. Constructing elliptic curve cryptosystems in characteristic 2. In *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, pages 156–167, 1990.

[14] Pierrick Gaudry. Fast genus 2 arithmetic based on theta functions. *J. Mathematical Cryptology*, 1(3):243–265, 2007.

[15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514, 2015.

[16] Joost Renes, Peter Schwabe, Benjamin Smith, and Lejla Batina. μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 301–320, 2016.

[17] Daniel J. Bernstein and Tanja Lange. Elliptic vs. hyperelliptic, part 1. *Talk at ECC*, page 4, 2006.

[18] Pierrick Gaudry and Éric Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.

[19] Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. Fast cryptography in genus 2. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 194–210, 2013.

[20] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer strikes back: New DH speed records. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, pages 317–337, 2014.

[21] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[22] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography, 2016.

[23] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. *Deep Space Network Progress Report*, 44:114–116, 1978.

[24] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of SIDH public keys. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 679–706, 2017.

[25] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Automatic generation of high-performance modular multipliers for arbitrary Mersenne primes on FPGAs. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*, pages 35–40, 2017.

[26] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. X25519 hardware implementation for low-latency applications. In *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*, pages 99–106, 2016.

[27] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Low-latency X25519 hardware implementation: Breaking the 100 microseconds barrier. *Microprocessors and Microsystems - Embedded Hardware Design*, 52:491–497, 2017.

[28] Philipp Koppermann, Fabrizio De Santis, Johann Heyszl, and Georg Sigl. Fast FPGA implementations of Diffie-Hellman on the Kummer surface of a genus-2 curve. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):1–17, 2018.

[29] Philipp Koppermann, Eduard Pop, Johann Heyszl, and Georg Sigl. 18 seconds to key exchange: Limitations of supersingular isogeny diffie-hellman on embedded devices. Cryptology ePrint Archive, Report 2018/932, 2018.

[30] Richard E. Crandall. Method and apparatus for public key exchange in a cryptographic system, 1992. US Patent 5,159,632.

[31] Craig Costello and Patrick Longa. Four$\mathbb{Q}$: Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, pages 214–235, 2015.

[32] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[33] Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.

[34] Gary W. Bewick. *Fast multiplication: algorithms and implementation.* PhD thesis, The Department of Electrical Engineering, Stanford University, 1994.

[35] Kimmo Järvinen, Andrea Miele, Reza Azarderakhsh, and Patrick Longa. Four$\mathbb{Q}$ on FPGA: New hardware speed records for elliptic curve cryptography over large prime characteristic fields. In *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, pages 517–537, 2016.

[36] Mark Hamilton, William P. Marnane, and Arnaud Tisserand. A comparison on FPGA of modular multipliers suitable for elliptic curve cryptography over GF(p) for specific p values. In *International Conference on Field Programmable Logic and Applications, FPL 2011, September 5-7, Chania, Crete, Greece*, pages 273–276, 2011.

[37] Pascal Sasdrich and Tim Güneysu. Implementing Curve25519 for side-channel-protected elliptic curve cryptography. *TRETS*, 9(1):3:1–3:15, 2015.

[38] Robert Granger and Michael Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In *Public-Key Cryptography - PKC 2015 - 18th IACR International Conference on Practice and Theory in Public-Key Cryptography, Gaithersburg, MD, USA, March 30 - April 1, 2015, Proceedings*, pages 539–553, 2015.

[39] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman and Hall/CRC, 2005.

[40] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, Berlin, Heidelberg, 2003.

[41] Joseph H.Silverman. *The arithmetic of elliptic curves*, volume 106. Springer Science & Business Media, 2 edition, 2009.

[42] Daniel J Bernstein. 25519 naming, 2014.

[43] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pages 320–339, 2012.

[44] Gesine Hinterwälder, Amir Moradi, Michael Hutter, Peter Schwabe, and Christof Paar. Full-size high-security ECC implementation on MSP430 microcontrollers. In *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers*, pages 31–47, 2014.

[45] Fabrizio De Santis, Omar Grati, Patrick Kresmer, Hermann Seuschek, and Georg Sigl. High-speed Curve25519 scalar multiplication on ARM Cortex-M4 microcontrollers. In Fachgruppe Kryptographie in der Gesellschaft für Informatik, editor, *23. Workshop der Fachgruppe Kryptographie in der Gesellschaft fü Informatik (Kryptotag)*, 2015.

[46] Michael Hutter, Jürgen Schilling, Peter Schwabe, and Wolfgang Wieser. Nacl's crypto_box in hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2015*, pages 81–101, 2015.

[47] Xilinx. Zynq-7000 all programmable SoC overview, 2016.

[48] Altera. Altera's user-customizable ARM-based SoC, 2015.

[49] Pascal Sasdrich and Tim Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In *Reconfigurable Computing: Architectures, Tools, and Applications - 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings*, pages 25–36, 2014.

[50] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302, 1999.

[51] Junfeng Fan and Ingrid Verbauwhede. An updated survey on secure ECC implementations: Attacks, countermeasures and cost. In *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, pages 265–282, 2012.

[52] Daniel J. Bernstein. 25519 naming. posting to the cfrg mailing list, 2014.

[53] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[54] Joppe W. Bos. Constant time modular inversion. *J. Cryptographic Engineering*, 4(4):275–281, 2014.

[55] Steven D. Galbraith. *Mathematics of Public Key Cryptography.* Cambridge University Press, 2012.

[56] Tanja Lange. Efficient arithmetic on genus 2 hyperelliptic curves over finite fields via explicit formulae. *IACR Cryptology ePrint Archive*, 2002:121, 2002.

[57] Rober Harley. Fast arithmetic on genus two curves. In *ACM Transactions in Embedded Computing Systems - TECS*, 2000.

[58] David V. Chudnovsky and Gregory V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Adv. Appl. Math.*, 7(4):385–434, 1986.

[59] Pierrick Gaudry. Variants of the montgomery form based on theta functions, 2006.

[60] Ping Ngai Chung, Craig Costello, and Benjamin Smith. Fast, uniform scalar multiplication for genus 2 Jacobians with fast Kummers. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pages 465–481, 2016.

[61] Florent de Dinechin and Bogdan Pasca. Large multipliers with fewer DSP blocks. In *19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic*, pages 250–255, 2009.

[62] Shuli Gao, Dhamin Al-Khalili, Noureddine Chabini, and J. M. Pierre Langlois. Asymmetric large size multipliers with optimised FPGA resource utilisation. *IET Computers & Digital Techniques*, 6(6):372–383, 2012.

[63] Edlyn Teske. An elliptic curve trapdoor system. *J. Cryptology*, 19(1):115–133, 2006.

[64] Alexander Rostovtsev, Elena Makhovenko, and Olga Shemyakina. Elliptic curve ordered digital signature. *Saint-Petersburg State Polytechnical University, April*, page 6, 2004.

[65] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Adv. in Math. of Comm.*, 4(2):215–235, 2010.

[66] Andrew M. Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Mathematical Cryptology*, 8(1):1–29, 2014.

[67] Steven D. Galbraith and Frederik Vercauteren. Computational problems in supersingular elliptic curve isogenies. *IACR Cryptology ePrint Archive*, 2017:774, 2017.

[68] Lawrence C. Washington. *Elliptic curves: number theory and cryptography.* CRC press, 2 edition, 2008.

[69] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Information Theory*, 39(5):1639–1646, 1993.

[70] Jacques Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris Sér. AB*, 273:A238–A241, 1971.

[71] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *J. Mathematical Cryptology*, 8(3):209–247, 2014.

[72] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 572–601, 2016.

[73] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IACR Cryptology ePrint Archive*, 2017:1015, 2017.

[74] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari Kermani. NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, pages 88–103, 2016.

[75] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Transactions on Computers*, pages 1–1, 2018.

[76] Reza Azarderakhsh, Brian Koziel, Seyed Hamed Fatemi Langroudi, and Mehran Mozaffari Kermani. FPGA-SIDH: High-performance implementation of supersingular isogeny Diffie-Hellman key-exchange protocol on FPGA. *IACR Cryptology ePrint Archive*, 2016:672, 2016.

[77] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Post-quantum cryptography on FPGA based on isogenies on elliptic curves. *IEEE Trans. on Circuits and Systems*, 64-I(1):86–99, 2017.

[78] Anatoly Karatsuba and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *Proc. of the USSR Academy of Sciences*, 145:293–294, 1962.

[79] Michael Scott. Fast machine code for modular multiplication, 1995.

[80] Zhe Liu and Johann Großschädl. New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers. In *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, pages 215–234, 2014.

[81] Amir Jalali, Reza Azarderakhsh, and Mehran Mozaffari Kermani. Efficient post-quantum undeniable signature on 64-bit ARM. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 281–298, 2017.

[82] Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. Newhope on ARM Cortex-M. In *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, pages 332–349, 2016.

[83] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1006–1018, 2016.

[84] Xinyu Lei and Xiaofeng Liao. NTRU-KE: A lattice-based public key exchange protocol. *IACR Cryptology ePrint Archive*, 2013:718, 2013.

[85] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In *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*, pages 63–91, 2016.

[86] Daniel Kirkwood, Bradley C. Lackey, John McVey, Mark Motley, Jerome A. Solinas, and David Tuller. Failure is not an option: Standardization issues for post-quantum key agreement. In *Talk at NIST workshop on Cybersecurity in a Post-Quantum World*, volume 2, 2015.

[87] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[88] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. Statistics and secret leakage. *ACM Trans. Embedded Comput. Syst.*, 3(3):492–508, 2004.

[89] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pages 495–513, 2015.