Doctoral Dissertation

A study on Compact Elliptic Curve Cryptosystems Resisting Side Channel Attacks

（サイドチャネル攻撃に安全な軽量な楕円曲線暗号の研究）

Graduate School of Engineering
Osaka University

Jin Yaoan

January 2023

Supervisor: Professor Atsuko Miyaji

Abstract

Today, internet of things (IoT) devices have penetrated into thousands of households. While enjoying the convenience of IoT devices, our interests are also compromised by the security vulnerabilities of IoT devices. The security of IoT devices should be taken seriously. In particular, the development of side channel attacks (SCAs) targeting IoT devices in recent years makes the security of IoT devices more and more important. Therefore, IoT devices with limited computational resources require compact cryptosystems resisting SCAs.

Elliptic curve cryptosystems (ECCs) are typical public key cryptosystems (PKCs) that can ensure equivalent security with considerably shorter keys than Rivest-Shamir-Adleman (RSA). Therefore, compact ECCs are recommended for IoT devices. In ECCs, elliptic curve scalar multiplications (ECSMs) are dominant computations. Therefore, enhancing the security and efficiency of ECSMs is important. In ECSMs, modular inversions are required when affine elliptic curve addition formulae are used. Modular inversions are also critical computations in elliptic curve digital signature algorithms (ECDSAs). Therefore, we also focus on secure and compact modular inversions.

An ECSM consists of a scalar multiplication algorithm and elliptic curve addition formulae. Elliptic curve addition formulae based on affine coordinates are memory-saving and can be efficient according to the ratio of modular inversion cost to modular multiplication cost but weak against SCAs. Elliptic curve complete addition (CA) formulae based on projective coordinates can achieve secure ECSMs resisting SCAs but are not compact. We focus on secure and efficient ECSMs taking advantage of the compact affine elliptic curve addition formulae.

Greatest common divisor (GCD) computations or modular inversions are used in RSA key generation, affine elliptic curve addition formulae, and ECDSAs. Therefore, we need focus on them for secure and efficient ECCs. GCD computations and modular inversions are often computed using the binary Euclidean algorithm (BEA) and binary extended Euclidean algorithm (BEEA), respectively. Therefore, the SCA weaknesses of BEA and BEEA become a serious concern. Constant-time GCD (CT-GCD) and constant-time modular inversion (CTMI) algorithms are effective countermeasures in such situations. A modular inversion by Fermat's little theorem (FLT) can work in constant time, but it is not efficient for general inputs. Two CTMI algorithms named BOS and BY, were proposed by Bos, Bernstein and Yang, respectively. Their algorithms are all based on the concept of BEA. However, one iteration of BOS has complicated computations, and BY requires many iterations. A small number of iterations and simple computations during one iteration are good characteristics of a constant-time algorithm.

In the first part of this thesis, three notions of a secure ECSM, named *generality of k*, *secure generality*, and *executable coordinates*, are proposed. Then, the original affine elliptic curve addition formulae are extended to eliminate some exceptional points. We improve Joye's regular 2-ary RL algorithm and propose new RL ECSMs, which can scan the input scalars from right to left (RL) using (extended) affine coordinates. For the security, we prove that our RL ECSMs satisfy secure generality and (extended) affine coordinates are executable coordinates for them.

In the second part of this thesis, new LR ECSMs scanning the input scalars from left to right (LR) are proposed, which are more memory-saving. Our LR ECSM with a memory of 12 field elements uses the least amount of memory. The secure generality of our LR ECSMs and that (extended) affine coordinates are executable coordinates for them are also proved. The efficiency of both LR and RL ECSMs is enhanced by optimizing the number of inversions. Compared with prior works, the efficiency of our LR (RL) ECSMs is analyzed from theoretical and experimental perspectives.

In the third part of this thesis, new short-iteration CT-GCD and CTMI algorithms over $\mathbb{F}_p$, named SICT-GCD and SICT-MI, respectively, are proposed borrowing a simple concept from BEA. SICT-GCD and SICT-MI are evaluated from a theoretical perspective. Compared with modular inversions by FLT, BOS, BY, and the improved version of BY, SICT-GCD and SICT-MI are experimentally demonstrated to be faster.

Acknowledgments

I would like to thank everyone who helped me complete my PhD.

First of all, I would like to thank my supervisor, Professor Atsuko Miyaji. I studied in Miyaji lab for five years, from a graduate school student to a PhD student. At first, I was a computer science undergraduate student and Professor Atsuko Miyaji gave me a lot of opportunities, such as lectures and seminars, to learn cryptography, information security. Professor Atsuko Miyaji also gave me a lot of research comments. She is my leader in the study of cryptography theory and applications.

Next, I would like to thank members in the elliptic curve cryptosystems team, Kenta Kodera, Mathieu de Goyon, Hayato Arai, Yusen Wang, and Yue Gao. We study and do research together and I really learned a lot from them. I also would like to thank Professor Kiyofumi Tanaka for helping my works about FPGA implementations, and Professor Tetsuya Takine, lecturer Yuntao Wang for reviewing my dissertation and giving me valuable feedbacks. Thanks Professor Akihiro Maruta, Professor Yuichi Tanaka, Professor Masakazu Sanbei, Professor Kyo Inoue, Professor Takashi Washio, and Professor Kazunori Komatani for their help in my study and PhD thesis. A very special thank goes to Professor Katsuyuki Takashima. He also took time to review my dissertation from his busy schedule and gave me valuable feedbacks.

Next, I am also very grateful to my best friend, the member in the same lab, Xiaolong Liu, for studying and playing with me for four happy years.

Finally I would like to thank my family for their support in my life and spirit. I appreciate my girlfriend, Jieni Liu for her company and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

With the development of technology, internet of things (IoT) devices have penetrated into thousands of households. While enjoying the convenience of IoT devices, our interests are also compromised by the security vulnerabilities of IoT devices. The security of IoT devices should be taken seriously. Therefore, IoT devices with limited computational resources require compact and efficient cryptosystems. In particular, the development of side channel attacks (SCAs) targeting IoT devices in recent years makes the security of IoT devices more and more important.

SCAs threaten cryptosystems that are mathematically proved to be secure and has been implemented on IoT devices or personal computers (PCs). SCAs make it possible that secret information can be obtained by analyzing various physical parameters including power consumption, electromagnetic emission, implementation time, and errors during the execution of cryptographic algorithms. In recent years, more than half of works on cryptographic hardware in international conferences (For example, the publications at the annual conference on cryptographic hardware and embedded systems (CHES)) are about SCAs. The endless emergence and improvement of SCA methods make such attacks increasingly dangerous [SRP02, YF14]. Therefore, SCAs cannot be ignored when considering the security of IoT devices.

Elliptic curve cryptosystems (ECCs) are typical public key cryptosystems (PKCs) that can guarantee sufficient security with smaller key sizes than Rivest-Shamir-Adleman (RSA). Hence, compact and efficient ECCs are recommended for blockchain applications [BHH+14, LMG+20] and IoT devices [AM11, NSS17]. Elliptic curve scalar multiplications (ECSMs) that compute kP for an elliptic curve point $P \in E(\mathbb{F}_q)$ and a scalar k are important computations frequently used in ECCs. Then, the efficiency of ECSMs is important. ECCs based on the hard problem named elliptic curve discrete logarithm problem (ECDLP: $\log_P(Q) = min\{k \in \mathbb{Z}_{\geq 0} | Q = kP\}$ for any elliptic curve points P and Q), where ECSMs are computed and the scalar k is a secret, make the security of ECSMs essential.

An ECSM consists of a scalar multiplication algorithm and elliptic curve addition formulae. Therefore, studies on secure and efficient ECSMs can be divided into two categories.

The first focuses on secure and efficient scalar multiplication algorithms [JY02, Joy07, Joy09, MM12, MMM04, EL09]. The second focuses on secure and efficient elliptic curve addition formulae [IT02, RCB16, Wro16, GJM+11, CMO98]. Studies on scalar multiplication algorithms aim to find efficient, regular, no-dummy and constant-time algorithms resisting SCAs. Elliptic curve addition formulae based on affine coordinates are memory-saving and can be efficient according to the ratio of modular inversion cost to modular multiplication cost but weak against SCAs. Elliptic curve complete addition (CA) formulae [RCB16, Wro16] can be used in ECSMs without introducing SCAs but are inefficient in terms of both memory and computational cost. Therefore, we focus on secure and efficient ECSMs taking advantage of the compact affine elliptic curve addition formulae. In ECSMs with affine elliptic curve addition formulae, modular inversions are used. Modular inversions are also the critical computations in elliptic curve digital signature algorithm (ECDSA). Therefore, the efficiency and security of modular inversions are important. The binary Euclidean algorithm (BEA) computes the greatest common divisor (GCD). It can be used in RSA key generation to check whether the GCD of the public key e and $\phi(n) = (p-1)(q-1)$, where p and q are randomly generated large primes, is equal to one. The binary extended Euclidean algorithm (BEEA) can be used in RSA key generation, affine elliptic curve addition formulae and ECDSA to compute a modular inversion. Both algorithms are attractive because they consist of only shift and subtraction computations. However, both BEA and BEEA, which are not regular and constant-time, have SCAs weaknesses. Specifically, the simple power analysis (SPA), the cache-timing attack (CTA), and the machine learning-based profiling attack (MLPA) were conducted on BEA and BEEA to recover secrets with a high success rate [ASSS17, AMSSS17, AGTB18, dlFPS+21, XLC+18]. Making algorithms constant-time is one of the countermeasures. Modular inversions based on Fermat's little theorem (FLT) can be computed in constant time. However, it is inefficient for general inputs and can only compute modular inversions over prime numbers. Two constant-time modular inversion (CTMI) algorithms based on the basic concept of BEA were proposed by Bos [Bos14], Bernstein and Yang [BY19], and are denoted by BOS and BY, respectively. The number of iterations of BY was improved by Pieter Wuille in [PGrb21], which is denoted by hdBY. However, BY (even hdBY) still requires more iterations than BOS, and the computations during one iteration of BOS are complicated. A small number of iterations and simple computations during one iteration are good characteristics of constant-time algorithms. Therefore, we focus on finding such CTMI algorithms.

1.2 Contributions

The contributions of this study can be divided into three parts. In the first part, 1) three notions of a secure ECSM, named *generality of k* (any scalar k can be the input of the ECSM), *secure generality* (the regular and no-dummy ECSM resisting SCAs with generality of k), and *executable coordinates* (coordinates with elliptic curve addition formulae used in the ECSM with secure generality, which do not introduce weaknesses of SCAs), are proposed. 2) Related works of RL scalar multiplication algorithms, Joye's double-add Alg. [Joy07] and Joye's regular 2-ary

RL Alg. [Joy09], are evaluated based on these three notions. The conclusion is that Joye's double-add algorithm is without secure generality and Joye's regular 2-ary RL algorithm is with secure generality. Affine coordinates are not executable coordinates for them. 3) Joye's regular 2-ary RL algorithm is improved to make affine coordinates be executable coordinates for it except $k = 0$. Then, the original affine elliptic curve addition formulae are extended to eliminate some exceptional computations. Improved Joye's regular 2-ary RL algorithm with (extended) affine elliptic curve addition formulae forms a new 2-ary RL ECSM with secure generality. 4) The new 2-ary RL ECSM is improved to the new two-bit 2-ary RL ECSM by scanning two bits of an input scalar each time and reducing the number of modular inversions. 5) The secure generality of our RL ECSMs and (extended) affine coordinates are executable coordinates for them are proved.

In the second part, we focus on LR ECSMs. The contributions are 1) related works of LR scalar multiplication algorithms, the Montgomery ladder [JY02] and Joye's regular 2-ary LR Alg. [Joy09], are evaluated based on the three notions. The conclusion is that the Montgomery ladder is without secure generality and Joye's regular 2-ary LR algorithm is with secure generality. Affine coordinates are not executable coordinates for them. 2) Using the Montgomery trick ($x_1^{-1} = x_2(x_1x_2)^{-1}$ and $x_2^{-1} = x_1(x_1x_2)^{-1}$ with only one modular inversion), eight affine combination-addition formulae using only one modular inversion are constructed. Applying affine combination-addition formulae in ECSMs can reduce the number of modular inversions and enhance the efficiency. 3) Joye's regular 2-ary LR algorithm is improved to make (extended) affine coordinates be executable coordinates for it. Then, the exception-free (EF) 2-ary LR ECSM using (extended) affine coordinates is proposed. The EF 4-ary LR ECSM using (extended) affine coordinates is proposed by scanning two bits of an input scalar each time and reducing the number of modular inversions. The EF 4-ary LR ECSM is extended to the extended exception-free (EX-EF) 4-ary LR ECSM to reduce modular inversions more. 5) Our LR ECSMs satisfy the secure generality and (extended) affine coordinates are executable coordinates for them are proved. 6) We review the RL ECSMs proposed in the first part and apply affine combination-addition algorithms to them to enhance efficiency and reduce modular inversions. The EX-EF 4-ary RL ECSM algorithm using less modular inversions is improved from two-bit 2-ary RL ECSM. 7) The memory and computational efficiency of Joye's regular 2-ary RL (LR) algorithm with CA formulae, the Montgomery ladder with CA formulae, and our RL (LR) ECSMs are evaluated from a theoretical and experimental point of view.

For more details of our ECSMs evaluation, memory is evaluated by counting the number of $\mathbb{F}_p$ elements used in an ECSM, including the prime number p, elliptic curve parameters, an input scalar, an input elliptic curve point, and the intermediate memories used in the scalar multiplication and affine addition formulae. It shows that the EF 2-ary LR ECSM with or without affine combination-addition formulae uses the least amount of memory, which are 12 field elements. The EF 2-ary LR ECSM using 12 field elements reduces that of the Montgomery ladder with CA formulae and Joye's LR with CA formulae by 36.84%. For computational efficiency, Joye's regular 2-ary RL (LR) algorithm with CA formulae, the Montgomery ladder with CA formulae, and our RL (LR) ECSMs are theoretically analyzed and implemented on the

national institute of standards and technology (NIST) elliptic curves of NIST-224, NIST-225, and NIST-384 using the GNU multiple precision arithmetic library (GMP 6.1.2). Experiments show that, for elliptic curves NIST-224, NIST-256, NIST-384, the EX-EF 4-ary LR ECSM with affine combination-addition formulae is the most efficient and saves 34.41%, 29.9%, and 25.17% of the computational time of Joye's LR with CA formulae, respectively.

In the third part, we turn our attention to the secure and efficient modular inversions, which are also important for secure and efficient ECCs. The contributions can be concluded as: 1) New iteration formula with only shift and subtraction operations for constant-time GCD (CT-GCD) and CTMI is defined in Definition 7.1. 2) The convergence of the iteration formula is shown in Theorem 7.1. It shows that the iteration formula converges to $a_n = \gcd(a_0, b_0)$ and $b_n = 0$, where $a_0, b_0 \in \mathbb{Z}$, $a_0 \geq b_0 \geq 0$, and a_0 or b_0 is odd. The required number of iterations of the iteration formula is shown in Theorem 7.2, which is $\texttt{bitlen}(a_0)+\texttt{bitlen}(b_0)$. 3) Based on these, new CT-GCD and CTMI algorithms over $\mathbb{F}_p$ with simple computations in an iteration and a small number of iterations, named short-iteration constant-time GCD (SICT-GCD) and short-iteration constant-time modular inversion (SICT-MI), respectively, are proposed. 4) Compared with modular inversions by FLT (FLT-CTMI), BOS, BY, and hdBY, SICT-GCD and SICT-MI are evaluated from a theoretical perspective. The number of iterations, the number of $\mathbb{F}_p$ computations in an iteration, and the maximum parallel data flow in an iteration of them are analyzed. It shows that SICT-GCD and SICT-MI require fewer iterations and are compact. The efficiency of the SICT-GCD and SICT-MI algorithms on $\mathbb{F}_p$, FLT-CTMI, BOS, BY, and hdBY are also evaluated by experiments. Moreover, considering that modular inversions are used in affine elliptic curve addition formulae, the efficiency of elliptic curve affine addition formulae using different modular inversion methods is evaluated. The results in Table 7.9 show that our SICT-GCD on $\mathbb{F}_p$ saves 7.44%, 13.78%, 16.67%, 18.45%, 24.77%, 27.05% and 28.3% of clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit GCD computations, respectively. The results in Table 7.10 show that SICT-MI on $\mathbb{F}_p$ saves

- 10.86%, 16.44%, 23.33%, 60.24%, 80.29% and 82.78% of clock cycles of FLT-CTMI on 224-, 384-, 511-, 1020-, 1790- and 2048-bit modular inversions respectively.
- 40.86%, 41.43%, 41.8%, 39.53%, 38.34%, 37.60% and 37.66% of clock cycles of BOS on 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit modular inversions respectively.
- 17.41%, 16.08%, 18.67%, 16.55%, 16.89%, 17.53% and 17.76% of clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit modular inversions respectively.

In summary, compact and efficient ECSMs scanning an input scalar from right to left or left to right taking advantage of the compact affine elliptic curve addition formulae are proposed. Our ECSMs can resist SCAs. Compact and efficient constant-time GCD and CTMI algorithms, named SICT-GCD and SICT-MI, are proposed. The efficiency of proposed ECSMs, SICT-GCD, and SICT-MI is evaluated from a theoretical and experimental point of view. The experiments show that they all have better efficiency.

1.3 Organization

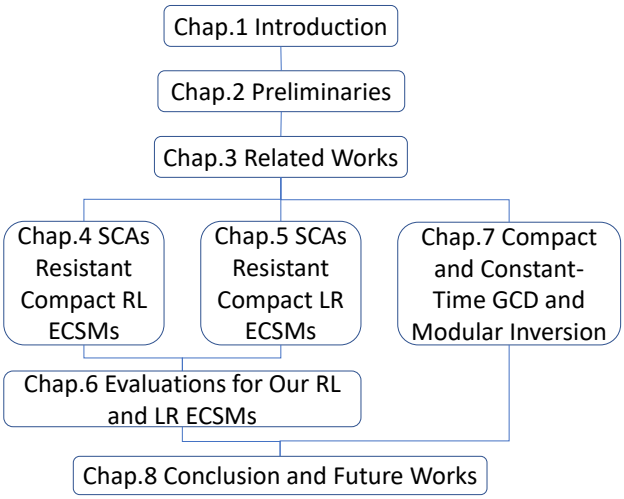

Figure 1.1: Chapter Relationship

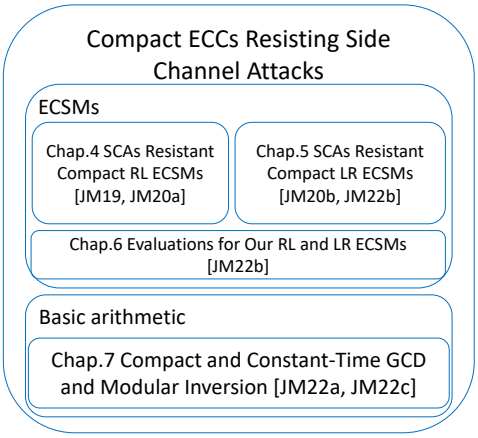

Figure 1.2: Proposal Relationship

The remainder of this study is organized as follows. Preliminary works are introduced in Chapter 2. Related works of secure and efficient ECSMs and modular inversions are introduced in Chapter 3. The study of RL ECSMs is presented in Chapter 4. The study of LR ECSMs is presented in Chapter 5. RL (LR) ECSMs are evaluated in Chapter 6. The study of CTMI algorithms is presented in Chapter 7. The summary and future work are shown in Chapter 8. The relationship among chapters is shown in Fig. 1.1 and the relationship among proposals is shown in Fig. 1.2. The relationships among the proposals and publications are:

- For the study of RL ECSMs, Chapter 4 is related to the journal 1 and international conference 2 in the list of publications, which are denoted by JM19 and JM20a, respectively.
- For the study of LR ECSMs, Chapter 5 is related to the journal 2 and international conference 3 in the list of publications, which are denoted by JM22b and JM20b, respectively.

- For the evaluations of RL and LR ECSMs, Chapter 6 is related to the journal 2 (JM22b) in the list of publications.
- For the study of CTMI, Chapter 7 is related to the journal 1 in the submitted for publication and international conference 5 in the list of publications, which are denoted by JM22c and JM22a, respectively.

which are also shown in Fig. 1.2.

Chapter 2

Preliminaries

This chapter introduces preliminary works that will appear in the next chapter. The algebraic structures, groups, rings, and fields, which are basic mathematical knowledge in cryptosystems, are introduced in Section 2.1. Two basic algorithms to compute scalar multiplications are introduced in Section 2.2. The greatest common divisor and modular inversions are important computations and frequently appear in cryptosystems. The basic methods for the greatest common divisor and modular inversions are introduced in Section 2.3. The efficiency of scalar multiplications, the greatest common divisor, and modular inversions is important for enhancing that of cryptosystems. Moreover, if these computations are related to some secret information, their security becomes critical. As the basic knowledge of ECCs, the theory of elliptic curve is introduced in Section 2.4. Compared to ECCs, another public-key cryptosystem named RSA is introduced in Section 2.5.

2.1 Basic Algebra

The basic abstract algebraic structures, groups, rings, and fields are the mathematical foundations of cryptosystems. This chapter will introduce the definitions of them.

2.1.1 Group

Definition 2.1. *A nonempty set $\mathbb{G}$ of any sort of elements (such as numbers, mappings, transformations) with a binary operation, denoted by $\cdot$, is said to be a group if the following four requirements are satisfied.*

1 *For any $a, b \in \mathbb{G}$, $a \cdot b \in \mathbb{G}$. The operation $\cdot$ can be any kind of operation, for example, addition in additive groups, multiplication in multiplicative groups.*

2 *The associative law: $\forall a, b, c \in \mathbb{G}$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*

3 *Identity element: There is a unique element $e \in \mathbb{G}$, called the identity element of the group, such that $\forall a \in \mathbb{G}$, $e \cdot a = a \cdot e = a$.*

4 Inverse element: $\forall a \in \mathbb{G}$, *there exists a unique element* $a^{-1} \in \mathbb{G}$, *such that* $a^{-1} \cdot a = a \cdot a^{-1} = e$. a^{-1} *is called the inverse of* a.

There are many examples of algebraic groups, such as $\mathbb{G} = \{0, 1, 2, 3, 4\}$ with the operation of modular multiplication by 5, $\mathbb{Z}$ with addition.

Remark 2.1. *An abelian group is with one more Characteristic that* $\forall a, b \in \mathbb{G}$, $a \cdot b = b \cdot a$ *(commutative law).*

Definition 2.2. *A nonempty subset* $H \subset \mathbb{G}$ *is said to be a subgroup of a given group* $\mathbb{G}$, *the following two conditions are necessary and sufficient.*

1 *For any* $a, b \in H$, $a \cdot b \in H$.

2 *For any* $a \in H$, $a^{-1} \in H$.

Equivalently, $\forall a, b \in H, a \cdot b^{-1} \in H$.

The number of elements of a group $\mathbb{G}$, $\#\mathbb{G}$, is called the *order* of the group $\mathbb{G}$. The *order* of an element $a \in \mathbb{G}$ is the least positive integer N, such that $a^N = e$. A group consisting of the powers of a single element is called a *cyclic group*. Then, we introduce Lagrange's theorem as following.

Theorem 2.1. *For any group* $\mathbb{G}$ *with order* N *and the order of every subgroup* $H \subset \mathbb{G}$, n, *then* $n|N$.

The groups with prime order do not consist of subgroups or elements with order two, because two cannot divides prime order according to Theorem 2.1.

2.1.2 Ring

Definition 2.3. *A set* R *with double operations, denoted by* $+$ *and* $\cdot$, *is called a* ring *if the following three requirements are satisfied:*

1 R *is an additive* $(+)$ *abelian group.*

2 R *is a monoid under multiplication* $(\cdot)$*:*

- *For any* $a, b \in R$, $a \cdot b \in R$.
- *The associative law:* $\forall a, b \in R$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- *There exists the identity element* e, *such that* $e \cdot a = a \cdot e = a$.

3 The distributive law: $\forall a, b, c \in R$, $a \cdot (b + c) = a \cdot b + a \cdot c$ *and* $(b + c) \cdot a = b \cdot a + c \cdot a$.

Remark 2.2. *A ring is commutative if the commutative law holds for multiplication* $(\cdot)$, $\forall a, b \in R$, $a \cdot b = b \cdot a$.

For example, $\mathbb{Z}$ with addition and multiplication is a typical commutative algebraic ring.

2.1.3 Field

Definition 2.4. *A field $\mathbb{F}$ is a commutative ring with multiplicative inverses that $\forall a \in \mathbb{F}$ and $a \neq e_0$, $\exists a^{-1} \in \mathbb{F}$, $a \cdot a^{-1} = a^{-1} \cdot a = e_1$, where e_0 is the identity element for addition and e_1 is the identity element for multiplication.*

The number of elements of a field $\#\mathbb{F}$ is called its *order*. A finite field with order q exists if and only if the order q is a prime power $p^k, k \in \mathbb{Z} > 0$ (where p is a prime number). If $q = p^k$, all fields of order q are isomorphic. In mathematics, an *isomorphism* is a structure-preserving mapping between two structures that can be reversed by an inverse mapping. Two mathematical structures are *isomorphic* if an isomorphism exists between them. Moreover, any subfield of a field with the same order must be the same. Therefore, all finite fields with the same order are unambiguously denoted by $\mathbb{F}_q$. For instance, the simplest finite fields with prime order are most directly accessible using modular arithmetic which can be $\mathbb{F}_n = \mathbb{Z}/n\mathbb{Z} = \{0, 1, \ldots, n-1\}$, where n is a prime number.

In mathematics, the *characteristic* of a ring R or a field $\mathbb{F}$, denoted by $\mathrm{char}(R)$ or $\mathrm{char}(\mathbb{F})$, is defined to be the smallest positive integer c such that $ce_1 = e_0$. If such equation does not exist, then the ring or field is said to have characteristic zero.

2.2 Scalar Multiplication

Scalar multiplications and modular scalar multiplications are basic and important computations that compute g^k, $g \in \mathbb{Z}$, $k \in \mathbb{Z} \geq 0$, and $g^k \mod p$, $g \in \mathbb{F}_p$, $k \in \mathbb{Z} \geq 0$, respectively. In this section, we introduce two basic methods to compute them, named binary method and double-and-add method, respectively. The difference between the binary method and the double-and-add method is that the binary method processes bits of a input scalar from left to right, and the double-and-add method processes bits of a input scalar from right to left.

2.2.1 Binary method

The binary method for computing modular scalar multiplications scanning bits of a input scalar from left to right is shown in Alg. 1. It computes a modular square in each iteration and a modular multiplication in the iteration where $k_i = 1$. Assuming that the most significant bit (MSB) of a input scalar is alway '1', then A can be initialized to g and the for loop starts from $i = \ell - 2$ to save one iteration.

2.2.2 Double-and-add method

The double-and-add method for computing modular scalar multiplications scanning bits of a input scalar from right to left is shown in Alg. 2. The basic concept of the double-and-add

Algorithm 1 Binary method

Input: $g \in \mathbb{F}_p$, $k \in \mathbb{Z} \geq 0$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$
Output: $A = g^k \mod p$

```
1: A ← 1;
2: for i = ℓ − 1 to 0 do
3:     A ← A^2 mod p;
4:     if k_i = 1 then
5:         A ← A · g mod p;
6:     end if
7: end for
8: return A;
```

method is shown in Equation (2.1).

$$g^k = g^{\sum_{i=0}^{\ell-1} k_i 2^i} = \prod_{i=0}^{\ell-1} g^{k_i 2^i}, k_i \in \{0, 1\} \tag{2.1}$$

The double-and-add method has the same computations as the binary method that computes a modular square in each iteration and a modular multiplication in the iteration, where $k_i = 1$. Rs with the form of g^{2^i} for i from 0 to $\ell - 1$ used when $k_i = 1$ can be precomputed and reused to enhance the efficiency of the double-and-add method.

Algorithm 2 Double-and-add method

Input: $g \in \mathbb{F}_p$, $k \in \mathbb{Z} \geq 0$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$
Output: $A = g^k \mod p$

```
1: A ← 1
2: R ← g
3: for i = 0 to ℓ − 1 do
4:     if k_i = 1 then
5:         A ← A · R mod p
6:     end if
7:     R ← R^2 mod p
8: end for
9: return A
```

The computation flow of Algs 1–2 highly depends on the inputs because they compute conditional statements or not according to every bit '1' or '0' of the input scalar k. The more bit '1' in the input scalar k, the more Execution time of Algs. 1–2.

2.3 Greatest Common Divisor (GCD) and Modular Inversion

Greatest common divisor (GCD) of a and b, denoted by $\gcd(a, b)$, and modular inversion of a over p, denoted by $a^{-1} \mod p$, are important computations used in cryptosystems. In this

section, we introduce two basic methods to compute a GCD, Euclidean algorithm and binary Euclidean algorithm, and three basic methods to compute a modular inversion, modular inversion based on Fermat's little theorem, extended Euclidean algorithm, and binary extended Euclidean algorithm.

2.3.1 Fermat's little theorem

Modular inversion methods can be divided into two categories. The first one is based on Fermat's little theorem (FLT), which is shown in Theorem 2.2. FLT implies that $a^{p-2} \cdot a \equiv 1 \mod p$, which is the basic idea for modular inversion based on FLT.

Theorem 2.2. *For any $a \in \mathbb{Z}$ and a prime p, $a^p - a \equiv 0 \mod p$.*

The modular inversion algorithm based on FLT, Alg 3, computes $a^{-1} \equiv a^{p-2} \mod p$ by the binary method in constant time. The computation flow for any input a over the same p is the same because the same prime p makes "if statements" be executed in the same way for any input a. The number of iterations is $\text{bitlen}(p-2) - 1$, where $\text{bitlen}(p-2)$ is the bit length of $p-2$. The efficiency of constant-time modular inversion based on FLT, denoted by FLT-CTMI, depends on the Hamming weight of $p-2$. The larger the Hamming weight of $p-2$, the lower the efficiency of FLT-CTMI. Thus, FLT-CTMI is inefficient for general inputs. Moreover, neither GCD nor modular inversion on a composite number can be computed based on FLT.

Algorithm 3 FLT-CTMI

Input: a, p, $a \in \mathbb{Z}$, $\gcd(a, p) = 1$, and p is a prime.
Output: $C = a^{-1} \mod p$.

1: $C = a$;
2: $k = p - 2$ $(\sum_{i=0}^{n-1} k_i 2^i)$;
3: **for** $i = n - 2$ to 0 **do**
4: $\quad C = C^2 \mod p$;
5: $\quad$ **if** $k_i = 1$ **then**
6: $\quad\quad C = C \cdot a \mod p$;
7: $\quad$ **end if**
8: **end for**
9: **return** C

2.3.2 (Extended) Euclidean algorithm

Another modular inversion is based on the idea of the extended Euclidean algorithm. Euclidean algorithm, Alg. 4, is used to compute GCD, whose basic concept is $\gcd(a, b) = \gcd(b, a \mod b)$ for $a, b \in \mathbb{Z}$. The transition matrices of Euclidean algorithm are defined in Definition 2.5. According to the transition matrices, $(\gcd(a_0, b_0), 0) = (a_0, b_0)T_0 \cdots T_n$ can be obtained. Using the equation of $a_0 x + b_0 y = \gcd(a_0, b_0) = 1$, the modular inversion of $b_0^{-1} \equiv y \mod a_0$ can be obtained, where x is the top-left element of $T_0 \cdots T_n$ and y is the bottom-left element of $T_0 \cdots T_n$. Thus, we have the extended Euclidean algorithm to compute modular inversions

shown in Alg. 5. The most notable feature of FLT-CTMI is that it can compute a modular inversion in constant time. By contrast, the extended Euclidean algorithm cannot compute a modular inversion in constant time. However, it is more efficient and can compute a modular inversion over both prime numbers and composite numbers.

Algorithm 4 Euclidean algorithm

Input: $a, b \in \mathbb{Z}$.
Output: $a = \gcd(a, b)$.
1: **while** $b \neq 0$ **do**
2: $r = a \mod b$;
3: $a = b$;
4: $b = r$;
5: **end while**
6: **return** a

Definition 2.5. *Transition matrices of Euclidean algorithm:*

$$T_n = \begin{pmatrix} 0 & 1 \\ 1 & -q_n \end{pmatrix}, (a_{n+1}, b_{n+1}) = (a_n, b_n)T_n, q_n : \textit{(Quotient of } a_n \textit{ divided by } b_n\textit{)}. \tag{2.2}$$

Algorithm 5 extended Euclidean algorithm

Input: $a, p \in \mathbb{Z}$, and $\gcd(a, p) = 1$.
Output: $y_1 = a^{-1} \mod p$.
1: $b = p$; $c = a$; $x_0 = 1$; $x_1 = 0$; $y_0 = 0$; $y_1 = 1$;
2: **while** $c \neq 1$ **do**
3: $r = b \bmod c$; $q = (b - r)/c$;
4: $b = c$; $c = r$;
5: $t = x_1$; $x_1 = x_0 - q \cdot x_1$; $x_0 = t$;
6: $t = y_1$; $y_1 = y_0 - q \cdot y_1$; $y_0 = t$;
7: **end while**
8: $y_1 = y_1 \bmod p$;
9: **return** y_1;

2.3.3 Binary (extended) Euclidean algorithm

Among all the variants of the Euclidean algorithm and extended Euclidean algorithm, the most frequently used variants are the binary Euclidean algorithm and binary extended Euclidean algorithm, which are called BEA and BEEA, respectively. BEA and BEEA consisting of only shift and subtraction operations, compute GCDs and modular inversions, respectively. They are attractive because shift and subtraction operations can more efficiently be implemented on

both the software and hardware. The basic concept of BEA and BEEA is as follows:

$$\gcd(a,b) = \begin{cases} \gcd(|a-b|, \texttt{min}(a,b)), & a \text{ and } b \text{ are odd} \\ \gcd(a/2, b), & a \text{ is even, } b \text{ is odd} \\ \gcd(a, b/2), & a \text{ is odd, } b \text{ is even} \\ 2 \cdot \gcd(a/2, b/2), & a \text{ and } b \text{ are even.} \end{cases} \tag{2.3}$$

BEA computes $\gcd(a,b)$ according to the Equation (2.3), as shown in Alg. 6. For modular inversion, the case that a and b are even does not exist, and the first "while" loop in Alg. 6 can be removed because $\gcd(a,p) = 1$ when $a^{-1} \mod p$ exists. The transition matrices of BEA are defined in Definition 2.6. Based on the transition matrices of BEA, BEEA is shown in Alg. 7 which computes the modular inversion of $a^{-1} \mod p$ ($\gcd(a,p) = 1$). Note that $>>$ indicates the right-shift operation and $<<$ indicates the left-shift operation. $A/2 \mod p$ can be computed as $A >> 1$ when A is even, and $(A+p) >> 1$ when A is odd.

Algorithm 6 BEA

Input: $a, b \in \mathbb{Z}$.
Output: $r = \gcd(a,b)$.

1: $u = a$; $v = b$; $r = 1$;
2: **while** u and v are even **do**
3: $\quad u = u >> 1$; $v = v >> 1$; $r = r << 1$;
4: **end while**
5: **while** $u \neq 0$ **do**
6: $\quad$ **while** u is even **do**
7: $\quad\quad u = u >> 1$;
8: $\quad$ **end while**
9: $\quad$ **while** v is even **do**
10: $\quad\quad v = v >> 1$;
11: $\quad$ **end while**
12: $\quad$ **if** $u \geq v$ **then**
13: $\quad\quad u = u - v$;
14: $\quad$ **else**
15: $\quad\quad v = v - u$;
16: $\quad$ **end if**
17: **end while**
18: $r = r \cdot v$;
19: **return** r

Definition 2.6. *Transition matrices of BEA:*

$$\begin{aligned}
T_n &= \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}, (a_{n+1}, b_{n+1}) = (a_n, b_n)T_n, \ a_n \text{ and } b_n \text{ are odd, and } a_n \geq b_n \\
T_n &= \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}, (a_{n+1}, b_{n+1}) = (a_n, b_n)T_n, \ a_n \text{ and } b_n \text{ are odd, and } a_n < b_n \\
T_n &= \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{pmatrix}, (a_{n+1}, b_{n+1}) = (a_n, b_n)T_n, \ a_n \text{ is even, } b_n \text{ is odd} \\
T_n &= \begin{pmatrix} 1 & 0 \\ 0 & \frac{1}{2} \end{pmatrix}, (a_{n+1}, b_{n+1}) = (a_n, b_n)T_n, \ a_n \text{ is odd, } b_n \text{ is even.}
\end{aligned} \tag{2.4}$$

Algorithm 7 BEEA

Input: a, p, where $\gcd(a, p) = 1$.

Output: $C = a^{-1} \mod p$.

1: $u = a$; $v = p$; $A = 1$; $C = 0$;
2: **while** $u \neq 0$ **do**
3: **while** u is even **do**
4: $u = u >> 1$; $A = A/2 \mod p$;
5: **end while**
6: **while** v is even **do**
7: $v = v >> 1$; $C = C/2 \mod p$;
8: **end while**
9: **if** $u \geq v$ **then**
10: $u = u - v$; $A = A - C$;
11: **else**
12: $v = v - u$; $C = C - A$;
13: **end if**
14: **end while**
15: $C = C \mod p$;
16: **return** C

2.4 Elliptic Curve

An elliptic curve over a field K, denoted by $E(K)$, is a projective curve of genus one with a specified K-rational point. The field K is usually taken to be the complex numbers, the reals, the rationals, the algebraic extensions of rationals, the p-adic numbers, or a finite field. Our elliptic curves are defined on a finite field. An elliptic curve of Weierstrass form over a field K with $char(K) \neq 2, 3$ is shown in Definition 2.7.

Definition 2.7. *(Weierstrass equation). An elliptic curve over a field K with $char(K) \neq 2, 3$ of an equation*

$$Y^2Z = X^3 + aXZ^2 + bZ^3, a, b \in K, \text{ and } 4a^3 + 27b^2 \neq 0 \tag{2.5}$$

is called a Weierstrass form elliptic curve. The point $\mathcal{O} = (0 : 1 : 0)$ is called the point at infinity of the elliptic curve. In affine, by defining the coordinates $x = X/Z$ and $y = Y/Z$, the equation can be rewritten as:

$$y^2 = x^3 + ax + b, a, b \in K, \text{ and } 4a^3 + 27b^2 \neq 0 \tag{2.6}$$

Note that the affine equation does not include the point $\mathcal{O} = (0 : 1 : 0)$, and an additional point at infinity $\mathcal{O}$ need to be added to it.

$D = 4a^3 + 27b^2 \neq 0$ is called the *elliptic discriminant*. $j = \frac{4.1728a^3}{D}$ is called the *j-invariant*. K-rational points of an elliptic curve ($E(K) = \{(x, y) \in K^2 | y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}$) with elliptic curve addition formulae form a group. The identity element of the elliptic curve is the point at infinity $\mathcal{O}$ and the inverse element of a point $E(K) \ni P = (x, y)$ is $-P = (x, -y)$.

The group consisting of all elliptic curve points P such that $nP = \mathcal{O}$ is the *n-torsion group* of $E(K)$, denoted by $E[n]$. For each point in this group, we say it is a *n-torsion point*. In other words, the order of the point is n.

2.4.1 Affine addition formulae

Different coordinates can be used for an elliptic curve. The different coordinates make the representation of points and addition formulae between two points different. In affine coordinates, elliptic curve points are represented by a pair of finite field elements (x, y) satisfying the Equation (2.6). The point at infinity $\mathcal{O}$ cannot be represented exactly in affine coordinates. Therefore, the affine addition formula 2.7 and affine doubling formula 2.8 define the affine addition operation between two points.

affine addition formula $(P \neq \pm Q)$

$$\begin{aligned} x_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \\ y_3 &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \end{aligned} \tag{2.7}$$

affine doubling formula

$$\begin{aligned} x_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \\ y_3 &= \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \end{aligned} \tag{2.8}$$

Note that the exceptional computations of $P + \mathcal{O}$, $P - P$, and $2\mathcal{O}$ are not included in Equations (2.7)–(2.8). Equations (2.7)–(2.8) with these three exceptional computations form a complete additive law over elliptic curves on affine coordinates.

2.4.2 Projective addition formulae

An elliptic curve can be represented by Projective coordinates. By replacing $x = X/Z$ and $y = Y/Z$ in Equation (2.6), an elliptic curve point is represented by three finite field elements

(X, Y, Z) satisfying the Equation (2.5). The Projective addition formula and doubling formula are shown in the Equation (2.9)–(2.10), respectively.

Projective addition formulae $(P \neq Q)$

$$\begin{cases} X_3 = vA \\ Y_3 = u(v^2 X_1 Z_2 - A) - v^3 Y_1 Z_2 \\ Z_3 = v^3 Z_1 Z_2 \end{cases} \tag{2.9}$$

$$u = Y_2 Z_1 - Y_1 Z_2,\ v = X_2 Z_1 - X_1 Z_2,$$
$$A = u^2 Z_1 Z_2 - v^3 - 2v^2 X_1 Z_2.$$

Projective doubling formulae

$$\begin{cases} X_3 = 2hs \\ Y_3 = w(4B - h) - 8Y_1^2 s^2 \\ Z_3 = 8s^3 \end{cases} \tag{2.10}$$

$$w = aZ_1^2 + 3X_1^2,\ s = Y_1 Z_1,$$
$$B = X_1 Y_1 s,\ h = w^2 - 8B.$$

The addition result of $P = (X, Y, 1)$ and $-P = (X, -Y, 1)$ by the equation (2.9) is $(0, 8Y^3, 0)$. $(0, 8Y^3, 0) \sim (0 : 1 : 0)$ is also the point at infinity. Thus, the Equation (2.9) can compute $P - P = \mathcal{O}$ correctly. Doubling of $\mathcal{O} = (0, 1, 0)$ by the equation (2.10) is $(0, 0, 0)$. $(0, 0, 0) \sim (0 : 1 : 0)$ is also the point at infinity. Thus, the Equation (2.10) can compute $2\mathcal{O}$ correctly. In summary, the exceptional computation of $P + \mathcal{O}$ and the equations (2.9)–(2.10) form a complete additive law over elliptic curves on Projective coordinates.

2.4.3 Jacobian addition formulae

By replacing $x = X/Z^2$, $y = Y/Z^3$ in Equation (2.6), an elliptic curve point is represented by the triple of (X, Y, Z) on Jacobian coordinates satisfying the Equation (2.11).

$$Y^2 = X^3 + aXZ^4 + bZ^6, a, b \in K, \text{ and } 4a^3 + 27b^2 \neq 0 \tag{2.11}$$

The point at infinity is represented as $\mathcal{O} = (1 : 1 : 0)$. Therefore, we have Jacobian addition formula, the Equation (2.12), and Jacobian doubling formula, the Equation (2.13).

Jacobian addition formulae $(P \neq Q)$

$$\begin{cases} X_3 = R^2 - J_1 - 2J_2 \\ Y_3 = R(J_2 - X_3) - 2S_1 J_1 \\ Z_3 = ((Z_1 + Z_2)^2 - Z_1^2 - Z_2^2)H \\ \quad\ = 2Z_1 Z_2 H \end{cases} \tag{2.12}$$

$$U_1 = X_1 Z_2^2,\ U_2 = X_2 Z_1^2,\ S_1 = Y_1 Z_2^3,$$
$$S_2 = Y_2 Z_1^3,\ H = U_2 - U_1,\ R = 2(S_2 - S1),$$
$$I = (2H)^2,\ J_1 = IH,\ J_2 = IU_1.$$

Jacobian doubling formulae

$$\begin{cases} X_3 = M^2 - 2S \\ Y_3 = M(S - X_3) - 8Y_1^4 \\ Z_3 = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2 = 2Y_1 Z_1 \end{cases} \tag{2.13}$$

$$S = 2((X_1 + Y_1^2)^2 - X_1^2 - Y_1^4) = 4X_1 Y_1^2,$$
$$M = 3X_1^2 + aZ_1^4.$$

The addition result of $P = (X, Y, 1)$ and $-P = (X, -Y, 1)$ by the Equation (2.12) is $(4Y^2, 8Y^3, 0)$. The affine coordinates $(4Y^2/0, 8Y^3/0) = (infinite, infinite)$, the point at infinity, can be obtained from $(4Y^2, 8Y^3, 0)$, although $(4Y^2, 8Y^3, 0)$ does not satisfy the Equation (2.11). Thus the Equation (2.12) can compute $P - P = \mathcal{O}$ correctly. Doubling of $\mathcal{O} = (1, 1, 0)$

by the Equation (2.13) is $(1, 1, 0)$, which is the point at infinity. Thus, the Equation (2.13) can compute $2\mathcal{O}$ correctly. In summary, the exceptional computation of $P + \mathcal{O}$ and the equations (2.12)–(2.13) form a complete additive law over elliptic curves on Jacobian coordinates.

2.4.4 Elliptic curve scalar multiplication (ECSM)

Elliptic curve scalar multiplication (ECSM) is an important and dominant computation in elliptic curve cryptosystems (ECC). ECSM computes kP with a point $P \in E(\mathbb{F}_p)$ and a scalar $k \in \mathbb{Z}$. ECSM consists of a scalar multiplication algorithm and elliptic curve addition formulae. Therefore, two simple ECSMs combining the binary method (Alg. 1) with elliptic curve addition formulae, and the double-and-add method (Alg. 2) with elliptic curve addition formulae are shown in Algs 8-9, respectively. In fact, an ECSM is much more complicated than what are shown in Algs.8-9. Because there exists exceptional points, which need additional conditional statement processing for both elliptic curve addition formula and elliptic curve doubling formula.

Algorithm 8 ECSM by binary method

Input: $P \in E(\mathbb{F}_p)$, $k \in \mathbb{Z} \geq 0$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$
Output: $A = kP$
1: $A \leftarrow \mathcal{O}$;
2: **for** $i = \ell - 1$ to 0 **do**
3: $\quad A \leftarrow 2A$;
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad A \leftarrow A + P$;
6: $\quad$ **end if**
7: **end for**
8: **return** A;

Algorithm 9 ECSM by double-and-add method

Input: $P \in E(\mathbb{F}_p)$, $k \in \mathbb{Z} \geq 0$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$
Output: $A = kP$
1: $A \leftarrow \mathcal{O}$;
2: $R \leftarrow P$;
3: **for** $i = 0$ to $\ell - 1$ **do**
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad A \leftarrow A + R$;
6: $\quad$ **end if**
7: $\quad R \leftarrow 2R$;
8: **end for**
9: **return** A;

2.4.5 Elliptic curve discrete logarithm problem

Definition 2.8. *Given $P, Q \in E(\mathbb{F}_p)$, elliptic curve discrete logarithm problem (ECDLP) is defined as:*

$$\log_P(Q) = \texttt{min}\{k \in \mathbb{Z}_{\geq 0} | Q = kP\}. \tag{2.14}$$

Elliptic curve discrete logarithm problem (ECDLP) shown in Definition 2.8 is a hard problem and one of the bases of ECC. Take elliptic-curve diffie–hellman (ECDH) as an example, which is a key agreement protocol. Base on the public information, an elliptic curve $E(\mathbb{F}_p)$, a point $P \in E(\mathbb{F}_p)$ with order N, Alice and Bob generate their own secret key $s_a \in [1, N-1]$ and $s_b \in [1, N-1]$ and compute $P_a = s_a P$ and $P_b = s_b P$, respectively. They exchange the public information P_a and P_b and compute $s_a P_b$ and $s_b P_a$, respectively. Finally, $Q = s_a s_b P$ is shared secretly because obtaining s_a (s_b) from P_a (P_b) and P is hard according to ECDLP. Therefore ECSMs became important computations in ECC and the input scalar k is often a secret. Enhancing the security and efficiency of ECSMs is essential.

2.4.6 Elliptic curve digital signature algorithm (ECDSA)

Elliptic curve digital signature algorithm (ECDSA) [JMV01] is a variant of the digital signature algorithm (DSA) based on ECDLP, which can be described as:

1. Key generation: based on the public information, an elliptic curve $E(\mathbb{F}_p)$, a point $P \in E(\mathbb{F}_p)$ with order N, a secret key $d \in [1, N-1]$ and a public key $Y = dP$ is generated.
2. Signature generation: a secret random number $k \in [1, N-1]$ is generated. Then the signature (r, s), $r = x(kP) \mod N \neq 0$, $s = k^{-1}(H(m) + rd) \mod N \neq 0$ is generated. $x(\cdot)$ returns the x-coordinate of an input point.
3. Signature verification: check if $x(\frac{H(m)}{s}P + \frac{r}{s}Y) \mod N = r$.

2.5 RSA (Rivest–Shamir–Adleman)

RSA (Rivest–Shamir–Adleman) [RSA78] is a classic public-key cryptosystem, which can be described as:

1. Key generation: RSA key generation in OpenSSL is shown in Alg. 10.

Algorithm 10 RSA key generation in OpenSSL

Input: A key size n and a public exponent e.

Output: Public and private key pair.

1: $p \leftarrow$ a $n/2$-bit random prime;
2: **while** $\gcd(p-1, e) \neq 1$ **do**
3: $\quad p \leftarrow$ a $n/2$-bit random prime;
4: **end while**
5: $q \leftarrow$ a $n/2$-bit random prime;
6: **while** $\gcd(q-1, e) \neq 1$ **do**
7: $\quad q \leftarrow$ a $n/2$-bit random prime;
8: **end while**
9: $N = pq$;
10: $d \leftarrow e^{-1} \mod (p-1)(q-1)$;
11: $dp \leftarrow d \mod (p-1)$;
12: $dq \leftarrow d \mod (q-1)$;
13: $iq \leftarrow q^{-1} \mod p$;
14: **return** (N, e), (d, p, q, dp, dq, iq);

2. Encryption: $c = m^e \mod N$.

3. Decryption: $c^d = m \mod N$.

Chapter 3

Related Works

3.1 Side Channel Attacks (SCAs)

Mathematically proved secure cryptosystems may have SCAs weaknesses after being implemented on IoT devices or PCs. SCAs analyze various physical signals including power consumption, electromagnetic emission, implementation time, and errors during the execution of cryptographic algorithms to reveal the secret information. Depending on the side channel information utilized, SCAs can be classified as simple power analysis (SPA) [Wal04], differential power analysis (DPA) [KJJ99], and template attack [SRP02] using power consumption or electromagnetic emission information; timing attack [Koc96] and cache-timing attack (CTA) [YF14] using implementation time information; safe error attack (SEA) [CKM21] and injection fault attack [KQ07] using errors information. For countermeasures,

- *Regular* algorithms that perform the same computations for each bit of the input resist SCAs using power consumption or electromagnetic emission information and CTA.
- *Constant-time* algorithms that perform the same computations for any input resist SCAs using power consumption, electromagnetic emission, or implementation time information.
- *No-dummy* algorithms where every statement is related to the final outputs resist SEA.

We can also hide the secret inputs by masking procedure [DFS19]. In recent years, more than half of the studies on cryptographic hardware are about SCAs. SCAs for GCD, modular inversions and ECSMs are introduced next.

GCD ($\gcd(a, b)$) and modular inversions ($a^{-1} \mod p$) are important computations frequently used in cryptosystems. GCD and modular inversions are often computed using the binary Euclidean algorithm (BEA) and binary extended Euclidean algorithm (BEEA), respectively. Therefore, the SCAs weaknesses of BEA and BEEA become a serious concern.

A method that can recover the inputs of BEA and BEEA was first proposed in [OSS07]. Specifically, the inputs of BEA and BEEA can be recovered by inputting the complete operational flow of an execution into the algorithm ([OSS07] Fig.7). Because u and v are always updated to zero and $\gcd(a, b)$ in the final, respectively, in Algorithms 6–7. With the complete

operational flow of an execution of BEA (or BEEA), we can inversely compute the initial values of u and v, which are the original inputs. The remaining question is how to obtain the complete operational flow of an execution of BEA (or BEEA).

The complete operational flow of an execution of BEA (or BEEA) can be obtained by SCAs. Although obtaining the complete operational flow by SCAs seems difficult, a lot of tricks make part of the operational flow enough to reveal the secret inputs of BEA and BEEA. Firstly, there are some "bad" characteristics in BEA and BEEA (Algorithms 6–7), which can be used to predict the operational flow: 1) there is at most one shift loop of either "u" or "v" in each iteration. 2) one subtraction must be performed in each iteration. Secondly, in practical applications, one of the inputs to BEA (or BEEA) is always public. Some characteristics of the inputs to BEA (or BEEA) can be known in advance. All these make SCAs to BEA and BEEA easier. Thirdly, the complete operational flow of an execution of BEEA was obtained by SPA and used to attack ECDSA in [ASSS17]. It is pointed out that the subtraction operational flow can be recovered from the shift operational flow. Thus, the complete shift operational flow of an execution of BEA (or BEEA) is sufficient to reveal the inputs.

Consider RSA key generation in OpenSSL (Alg. 10) as an example. $\gcd(e, p-1)$ and $\gcd(e, q-1)$, where e is a stable public key and p and q are random large prime numbers with the same bit length, are computed using BEA to check whether p and q are generated properly. The modular inversion of $d = e^{-1} \bmod (p-1)(q-1)$ is computed to obtain the secret key using BEEA. Based on the characteristics of RSA key generation: 1) e and $n = pq$ are public; 2) $(p-1)(q-1)$ has almost half the same most significant bits (MSBs) as n, the secret $(p-1)(q-1)$ can be recovered by a part of the operational flow at the beginning of the modular inversion computation, $e^{-1} \bmod (p-1)(q-1)$. Referring to n, it is sufficient to recover some least significant bits (LSBs) of $(p-1)(q-1)$ with the operational flow at the beginning and public input e [AMSSS17, AGTB18]. Benefiting from 1) the general setting of $e = 65537$ is much smaller than $(p-1)(q-1)$, $p-1$, and $q-1$; 2) $4|(p-1)(q-1)$, $2|(p-1)$, and $2|(q-1)$, the partial operational flow at the beginning of BEA (or BEEA) can be predicted in advance. Using all these characteristics, SPA, CTA, and machine learning-based profiling attack [SRP02] were conducted on RSA key generation [AMSSS17, AGTB18, dlFPS+21].

For countermeasures, one can hide the secret inputs of BEA and BEEA using appropriate masking procedures. The security of masking procedures was reported in [DFS19]. Another option is to use CT-GCD or CTMI.

SCAs have several attack methods for revealing bits of secret input scalars of ECSMs: SPA, which uses conditional statements in ECSMs to attack, and SEA, which uses dummy statements in ECSMs to attack [CJ03, FGMN16]. Therefore, to resist SPA and SEA, we must eliminate conditional statements and dummy statements in ECSMs. Regular and no-dummy ECSMs are required.

3.2 Constant-Time GCD, Modular Inversion and Montgomery Inversion

A Modular inversion can be computed in constant time by FLT, which is shown in Algorithm 3. Algorithm 3 is a CTMI, because p is the same for each modular inversion of a, and conditional statements are performed in the same way for any input a. The number of iterations is fixed to $\texttt{bitlen}(p-2)-1$, where $\texttt{bitlen}(p-2)$ is the bit length of $p-2$. The computations in each iteration (a modular squaring or a modular squaring with a modular multiplication), which depend on each bit of $p-2$, are more expensive than shift or subtraction and identical for any input a. The efficiency of CTMI by FLT, named FLT-CTMI, depends on the Hamming weight of $p-2$. The larger the Hamming weight of $p-2$, the lower the efficiency of FLT-CTMI. Thus, FLT-CTMI is inefficient for general inputs. Moreover, FLT can compute neither GCD nor modular inversions over composite numbers, which is required in RSA key generation, such as $\gcd(e, p-1)$ and $e^{-1} \mod (p-1)(q-1)$.

Kaliski's algorithm (Alg. 11) was introduced to compute Montgomery inversions ($a \cdot R \mod p$ where $R > p$ always has the form of 2^k, $k \in \mathbb{Z}$) [Kal95]. Bos proposed a CTMI by improving Kaliski's algorithm to a constant-time version, named BOS [Bos14]. The basic idea is to compute computations in four branches and then select the correct values from them according to the defined signal variables. The number of iterations of BOS is fixed as $2 \cdot \texttt{bitlen}(p)$, because the sum of the bit lengths of u and v is reduced by one in each iteration. BOS can compute Montgomery inversions, modular inversions by precomputing $2^{-2 \cdot \texttt{bitlen}(p)} \mod p$ and multiplying it to the final result, and $\gcd(a, b)$, where a or b is odd.

Algorithm 11 Kaliski's algorithm

Input: $a, p \in \mathbb{Z}$, $\gcd(a, p) = 1$, and $0 \leq a < p$.
Output: $s = a^{-1} \cdot 2^k \mod p$, and k

1: $u = p$; $v = a$; $r = 0$; $s = 1$; $k = 0$;
2: **while** $v \neq 1$ **do**
3: **if** u is even **then**
4: $u = u >> 2$; $s = s << 2$;
5: **else if** v is even **then**
6: $v = v >> 2$; $r = r << 2$;
7: **else if** $u > v$ **then**
8: $u = (u - v) >> 2$; $r = r + s$; $s = s << 2$;
9: **else**
10: $v = (v - u) >> 2$; $s = r + s$; $r = r << 2$;
11: **end if**
12: $k = k + 1$;
13: **end while**
14: **return** (s, k)

A CTMI was proposed by inserting dummy computations into the algorithm [SC21]. The number of iterations is fixed as $2 \cdot \texttt{bitlen}(p)$, the same as that of BOS. However, inserting dummy computations lowers the efficiency and creates potential SCAs security issues, such as

SEA.

Algorithm 12 BY [BY19]

Input: a, p, $l = \lfloor (49 \cdot \texttt{bitlen}(p) + 57)/17 \rfloor$, $pre\_com = 2^{-l} \bmod p$.
Output: $q = a^{-1} \bmod p$.

1: $u = a$; $v = p$; $q = 0$; $r = 1$; $\delta = 1$;
2: **for** $i = 0$ to $l - 1$ **do**
3: $\quad z = u_{lsb}$; $s = \texttt{signbit}(-\delta)$; $\delta = 1 + (1 - 2sz)\delta$;
4: $\quad u, v = (u + (1 - 2sz)zv) >> 1, v \oplus sz(v \oplus u)$;
5: $\quad q, r = (q \oplus sz(q \oplus r)) << 1, (1 - 2sz)zq + r$;
6: **end for**
7: $q = \texttt{sign}(v) \cdot q$;
8: $q = q \cdot pre\_com \bmod p$;
9: **return** q;

Bernstein and Yang proposed a CTMI algorithm on $\mathbb{F}_p$, Algorithm 12, named BY [BY19]. In Algorithm 12, `signbit(a)` returns 0 when $a \geq 0$ or 1 when $a < 0$, and `sign(a)` returns -1 when $a < 0$ or 1 when $a > 0$. The iteration formula of their algorithm is shown in Equation (3.1), where the input $b \in \mathbb{Z}$ should be odd.

$$F(\delta, b, a) = \begin{cases} (1 - \delta, a, \frac{a-b}{2}) & \delta > 0 \text{ and } a \text{ is odd} \\ (1 + \delta, b, \frac{a + (a \bmod 2)b}{2}) & \text{otherwise.} \end{cases} \quad (\delta, a, b \in \mathbb{Z}, b \text{ is odd.}) \tag{3.1}$$

The computations of their algorithm during one iteration are simpler than that of BOS. Bernstein and Yang analyzed the rate of shrinking of the transition matrix after each iteration and proved that the required number of iterations of BY is $\lfloor (49d + 57)/17 \rfloor (d \geq 46)$, where d is the largest bit length of the inputs. The number of iterations is significantly greater than that of BOS. BY can compute the modular inversions, Montgomery inversions, and $\gcd(a, b)$, where a or b is odd. The first k iterations of the iteration formulae of BY are completely determined by the lowest k bits of b and a. This feature supports the divide-and-conquer strategy of BY.

Bernstein and Yang also proposed a CTMI algorithm on $\mathbb{F}_{p^n}$. $\mathbb{F}_{p^n}$ is defined as $\mathbb{F}_p[x]/(f(x))$ and any element in $\mathbb{F}_{p^n}$ is represented by $g(x) \in \mathbb{F}_p[x]$. The iteration formula is given by Equation (3.2), where $f(0) \neq 0$.

$$F(\delta, f, g) = \begin{cases} (1 - \delta, g, (g(0)f - f(0)g)/x) & \delta > 0,\ g(0) \neq 0 \\ (1 + \delta, f, (f(0)g - g(0)f)/x) & \text{otherwise.} \end{cases} \quad \delta \in \mathbb{Z}, f(x), g(x) \in \mathbb{F}_p[x], f(0) \neq 0. \tag{3.2}$$

Each branch has computations of one addition (or subtraction) on $\mathbb{Z}$, one subtraction on $\mathbb{F}_{p^n}$, one shift on $\mathbb{F}_{p^n}$, and two multiplications on $\mathbb{F}_{p^n}$. The number of iterations is fixed at $2 \cdot \texttt{max}(\texttt{deg}(f), \texttt{deg}(g))$.

Pieter Wuille kept track of the convex hulls of possible (b, a) after each iteration to find the required number of iterations of BY and obtained similar results [PGrb21]. He showed that the required number of iterations of BY for 224-, 256-, and 384-bit computations are 634, 724, and 1086, respectively, whereas Bernstein and Yang showed they are 649, 741, and 1110,

respectively. Moreover, Pieter Wuille proposed a variant of BY, named hdBY, by initializing $\delta = 1/2$ or using the iteration formula in Equation (3.3).

$$F(\delta, b, a) = \begin{cases} (2-\delta, a, \frac{a-b}{2}) & \delta > 0 \text{ and } a \text{ is odd} \\ (2+\delta, b, \frac{a+(a \bmod 2)b}{2}) & \text{otherwise.} \end{cases} \quad (\delta, a, b \in \mathbb{Z}, b \text{ is odd.}) \tag{3.3}$$

The required number of iterations of hdBY, defined by $\texttt{max}(2\lfloor(2455\log_2(M) + 1402)/1736\rfloor, 2\lfloor(2455\log_2(M) + 1676)/1736\rfloor - 1)$, and $M \geq 157$, $0 \leq a \leq b \leq M$, is smaller. The necessary number of iterations of hdBY for 224-, 256-, and 384-bit computations are 517, 590, and 885, respectively.

3.3 Regular Scalar Multiplication Algorithms

Joye's double-add algorithm (Algorithm 13) is a regular scalar multiplication algorithm, which scans a scalar from right to left [Joy07]. Joye's regular 2-ary RL algorithm (Algorithm 15) by setting $m = 2$ in Joye's regular m-ary RL algorithm (Algorithm 14) is a regular and no-dummy scalar multiplication algorithm, also scanning a scalar from right to left [Joy09]. Therefore, the security of ECSMs using Joye's regular 2-ary RL algorithm depends on the coordinates with elliptic curve addition formulae. When using elliptic curve addition formulae on affine coordinates, conditional statements to process exceptional additions ($P+P$, $P-P$, and $\mathcal{O}+P$ for affine addition formula, and $2P = \mathcal{O}$ for affine doubling formula) are required. Conditional statements introduce SCAs weaknesses into ECSMs.

Algorithm 13 Joye's double-add algorithm [Joy07]

Input: $P \in E(\mathbb{F}_p), k = \sum_{i=0}^{\ell-1} k_i 2^i$.
Output: kP.
Uses: $R[0]$, $R[1]$.
1: $R[0] \leftarrow \mathcal{O}$;
2: $R[1] \leftarrow P$;
3: **for** $i = 0$ to $\ell - 1$ **do**
4: $\quad R[1-k_i] \leftarrow 2R[1-k_i] + R[k_i]$;
5: **end for**
6: **return** $R[0]$;

Three regular LR scalar multiplication algorithms, the Montgomery ladder [JY02], Joye's regular m-ary LR algorithm [Joy09], and Joye's regular 2-ary LR algorithm [Joy09] are shown in Algorithms 16–18, respectively. Joye's regular m-ary LR algorithm and Joye's regular 2-ary LR algorithm are also no-dummy scalar multiplication algorithms. Therefore, to construct secure ECSMs with them, coordinates with elliptic curve addition formulae should be taken carefully.

Algorithm 14 Joye's regular m-ary RL algorithm [Joy09]

Input: $P \in E(\mathbb{F}_p)$, $k = \sum_{i=0}^{\ell-1} k_i m^i$.
Output: kP.
Uses: A and $R[1]$, ..., $R[m]$.
Initialization
1: **for** $i = 1$ to m **do**
2: $\quad R[i] \leftarrow \mathcal{O}$;
3: **end for**
4: $A \leftarrow P$;
Main loop
5: **for** $i = 0$ to $\ell - 2$ **do**
6: $\quad R[1 + k_i] \leftarrow R[1 + k_i] + A$;
7: $\quad A \leftarrow mA$;
8: **end for**
Aggregation and final correction
9: $A \leftarrow (k_{\ell-1} - 1)A + \sum_{i=1}^{m}(m + i - 2)R[i]$;
10: $A \leftarrow A + P$;
11: **return** A;

3.4 Complete Addition (CA) Formulae

$$\begin{aligned} X_3 =& (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \\ & - (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) \\ Y_3 =& (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) \\ & + (Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) \\ Z_3 =& (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) \\ & + (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2). \end{aligned} \tag{3.4}$$

The elliptic curve complete addition formulae on projective coordinates can be used to compute the addition of any two elliptic curve points, $P, Q \in E(\mathbb{F}_p)$, $P = (X_1 : Y_1 : Z_1)$, $Q = (X_2 : Y_2 : Z_2)$, and $P + Q = (X_3 : Y_3 : Z_3)$ [RCB16], which are shown in Equation (3.4). Therefore, they can be combined with secure scalar multiplication algorithms without introducing conditional statements. For example, CA formulae are combined with the Montgomery ladder in [MLRB20]. However, CA formulae are inefficient in terms of both memory usage and computational costs. In addition, they only work for prime-order elliptic curves.

The Table 3.1 summarizes exceptional computations for each elliptic curve addition formulae. The Table 3.2 presents a comparison of the computational costs and memory usage for various elliptic curve addition formulae. The costs for a field multiplication (M), square (S), inversion (I), and addition (A) are considered, as well as the costs for a field multiplication with elliptic curve parameters a and b (m_a and m_b). When assuming $S = M$ and neglecting m_a, m_b, and A, the computational cost for an elliptic curve addition (ADD) and doubling (DBL) using the CA formulae is $24M$. The computational cost for an ADD and DBL using affine addition formulae is more efficient than those using the CA, Jacobian addition, or projective addition formulae

Algorithm 15 Joye's regular 2-ary RL algorithm [Joy09]

Input: $P \in E(\mathbb{F}_p), k = \sum_{i=0}^{\ell-1} k_i 2^i$.
Output: kP.
Uses: A, $R[0]$, $R[1]$.
Initialization
1: $R[0] \leftarrow \mathcal{O}$; $R[1] \leftarrow \mathcal{O}$; $A \leftarrow P$;
Main loop
2: **for** $i = 0$ to $\ell - 2$ **do**
3: $\quad R[k_i] \leftarrow R[k_i] + A$; $A \leftarrow 2A$;
4: **end for**
Aggregation and final correction
5: $A \leftarrow (k_{\ell-1} - 1)A + R[0] + 2R[1]$;
6: $A \leftarrow A + P$;
7: **return** A;

Algorithm 16 Montgomery ladder [JY02]

Input: $P \in E(\mathbb{F}_p)$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$.
Output: kP.
Uses: $R[0]$, $R[1]$.
1: $R[0] \leftarrow \mathcal{O}$, $R[1] \leftarrow P$;
2: **for** $i = \ell - 1$ to 0 **do**
3: $\quad R[1 - k_i] \leftarrow R[0] + R[1]$;
4: $\quad R[k_i] \leftarrow 2R[k_i]$;
5: **end for**
6: **return** $R[0]$;

when $I < 8.5M$, $I < 9M$, or $I < 9.5M$, respectively. When applied to NIST elliptic curves with $a = -3$ and a nonnegligible m_b, the total computational cost for an ADD and DBL using the CA formulae is $28M$. The computational cost for an ADD and DBL using affine addition formulae is much lower when $I < 10.5M$. The affine elliptic curve addition formulae using 5 field elements are memory-saving and optimal when considering the ratio of the modular inversion cost to the modular multiplication cost.

Table 3.1: Exceptional computations for elliptic curve addition formulae

Addition formulae	ADD formula	DBL formula
CA [RCB16]	-	-
Affine	$P + P$,$P - P$,$P + \mathcal{O}$	$2P = \mathcal{O}$
Jacobian [LM08]	$P + P$,$P + \mathcal{O}$	-
Projective [CMO98]	$P + P$,$P + \mathcal{O}$	-

Algorithm 17 Joye's regular m-ary LR algorithm [Joy09]

Input: $P \in E(\mathbb{F}_p)$, $k = \sum_{i=0}^{\ell-1} k_i m^i$.
Output: kP.
Uses: A and $R[1]$, ..., $R[m]$.
Initialization
1: **for** $i = 1$ to m **do**
2: $\quad R[i] \leftarrow (m + i - 2)P$;
3: **end for**
4: $A \leftarrow (k_{\ell-1} - 1)P$;
Main loop
5: **for** $i = \ell - 2$ to 0 **do**
6: $\quad A \leftarrow mA + R[1 + k_i]$;
7: **end for**
Final correction
8: $A \leftarrow A + P$;
9: **return** A;

Algorithm 18 Joye's regular 2-ary LR algorithm [Joy09]

Input: $P \in E(\mathbb{F}_p)$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$.
Output: kP.
Uses: A, $R[0]$, $R[1]$.
Initialization
1: $R[0] \leftarrow P$; $R[1] \leftarrow 2P$;
2: $A \leftarrow (k_{\ell-1} - 1)P$;
Main Loop
3: **for** $i = \ell - 2$ to 0 **do**
4: $\quad A \leftarrow 2A + R[k_i]$;
5: **end for**
Final Correction
6: $A \leftarrow A + R[0]$;
7: **return** A

Table 3.2: Comparison of elliptic curve addition formulae

Addition formulae	Conditions	ADD $(P + Q)$	DBL $(2P)$	Memory
CA [RCB16]	$2 \nmid \#E(\mathbb{F}_p)$	$12M + 3m_a + 2m_b + 23A$	$12M + 3m_a + 2m_b + 23A$	15
Affine	-	$2M + S + I + 6A$	$2M + 2S + I + 6A$	5
Jacobian [LM08]	-	$11M + 5S + 9A$	$M + m_a + 8S + 10A$	8
Projective [CMO98]	-	$12M + 2S + 6A$	$7M + m_a + 5S + 4A$	7

Chapter 4

SCAs Resistant and Compact RL ECSMs

In this chapter, we consider SCAs resistant and compact RL ECSMs. We first define three notions of SCAs resistant ECSMs, *Generality of k*, *Secure generality*, and *Executable coordinates* in Section 4.1. Then, six related works of scalar multiplication algorithms are analyzed from the three notions, and thus, we focus on Joye's regular 2-ary algorithm, which satisfies the *Secure generality* in Section 4.2. Next we extend elliptic curve affine addition formulae to eliminate some exception inputs in Section 4.3. Finally, based on three notions of SCAs resistant ECSMs, we improve Joye's regular 2-ary RL algorithm to make sure (extended) affine addition formulae can be used without introducing SCA weaknesses and propose our SCAs resistant RL ECSMs taking advantage of compact of affine addition formulae in Section 4.4.

4.1 Three Notions of Secure ECSMs

Three notions for secure ECSMs are defined as follows:

Definition 4.1. *Generality of k: An ECSM is of the generality of k if it can compute kP for an elliptic point $P \in E(\mathbb{F}_p)$ and a scalar $\forall k \in \mathbb{Z}/N\mathbb{Z}$ with $k_{\ell-1} = 0$ or $k_{\ell-1} = 1$, where $k \in \{0,1\}^\ell$ and N is the order of P.*

Definition 4.2. *Secure generality: An ECSM is of the secure generality if it is regular and no-dummy satisfying the generality of k.*

Definition 4.3. *Executable coordinates: Coordinates with elliptic curve addition formulae are executable for an ECSM, if they can be combined with a secure scalar multiplication algorithm without introducing exceptional computation cases. Note that algorithms with exceptional computation cases are not regular.*

4.2 Scalar Multiplication Algorithms Analysis

Based on the three notions of secure ECSMs, six related works of scalar multiplication algorithms (Algorithms 13–18) with an input scalar $k = \sum_{i=0}^{\ell-1} k_i 2^i$ and an elliptic curve point P are analyzed.

Theorem 4.1. *Joye's double-add algorithm (Algorithm 13) satisfies the* Generality of k. *The projective coordinates with CA formulae are its* Executable coordinates. *The affine and Jacobian coordinates are not the* Executable coordinates *of it.*

Proof.

1 *Generality of k*: Evidently, Algorithm 13 can compute kP correctly when $k \in \mathbb{Z}/N\mathbb{Z}$ and $k_{\ell-1} = 1$.

Algorithm 13 can also compute kP correctly when $k \in \mathbb{Z}/N\mathbb{Z}$ and $k_{\ell-1} = 0$ because it scans an input scalar from right and reads '0's at the end when $k_{\ell-1} = 0$ and the '0's read at the end do not change the value saved in $R[0]$, which is the final correct result. Therefore, Joye's double-add algorithm can compute for any input scalar $k \in \mathbb{Z}/N\mathbb{Z}$ with $k_{\ell-1} = 0$ or $k_{\ell-1} = 1$. Joye's double-add algorithm is of the *Generality of k*.

2 *Secure generality*: Algorithm 13 being able to compute regularly means that the computations for every bit of an input scalar are the same. However, it contains dummy statements. If an input scalar k starts with several '0's, Step 4 updating only $R[1]$ at the end of Algorithm 13 becomes a dummy computation. We can learn whether an input scalar k starts with several '0's via an SEA on Step 4. Therefore, Joye's double-add algorithm does not satisfy *Secure generality* and is not a secure scalar multiplication algorithm against SCA.

3 *Executable coordinates*: Algorithm 13 computes $R[1] \leftarrow 2P + \mathcal{O}$ when $k_0 = 0$ and $R[0] \leftarrow 2\mathcal{O} + P$ when $k_0 = 1$, which cannot be computed by affine or Jacobian coordinates. Evidently, the projective coordinates with CA formulae are *Executable coordinates* for it.

□

Lemma 4.1. *Joye's regular m-ary RL algorithm, Alg. 14, can compute kP correctly with any input scalar* $k \in \{0, 1, ..., m-1\}^\ell$ *and* $P \in E(\mathbb{F}_p)$.

Proof. Let $k = \sum_{i=0}^{\ell-1} k_i m^i$ be an m-ary representation of an input scalar with $k_i \in [0, m-1]$.

Case 1: The MSB of k is not zero, $k_{\ell-1} \in [1, m-1]$. Joye's regular RL m-ary algorithm, which computes kP correctly, is described in [Joy09].

Case 2: The MSB of k is zero, $k_{\ell-1} = 0$ and $k \neq 0$. Assuming that the number of '0's before the first bit k_i ($k_i \neq 0$) is N, the number of the rest bits is n. By Algorithm 14, the values are updated as:

	$R[1]$	$R[k_i+1]$	A
Initialization	$\mathcal{O}$	$\mathcal{O}$	P
Reading k_i	$\cdots$	$\cdots + m^{n-1}P$	m^nP
Reading the first '0' before k_i	$\cdots + m^nP$	$\cdots + m^{n-1}P$	$m^{n+1}P$
Reading $k_{\ell-2} = 0$	$\cdots + m^nP + \cdots + m^{N+n-1}P$	$\cdots + m^{n-1}P$	$m^{N+n}P$

Because of the '0's before k_i, in the aggregation of Algorithm 14, $(m^nP+\cdots+m^{N+n-1}P)(m-1) - m^{N+n}P = -m^nP$ is added as a part of the final result. Because of k_i, in the aggregation of Algorithm 14, $(m + k_i - 1)m^{n-1}P$ is added as a part of the final result. Removing the '0's before k_i, the final result should be the same and $(k_i - 1)m^{n-1}P$ is added as a part of the final result in the aggregation of Algorithm 14. Then, $-m^nP + (m + k_i - 1)m^{n-1}P = (k_i - 1)m^{n-1}P$ indicates that there's no influence to the final result when the MSB of scalar k is '0'.

Case 3: $k = 0$, $(P + mP + ... + m^{\ell-2}P)(m-1) - m^{\ell-1}P + P = O$.

Therefore, Joye's regular m-ary RL algorithm is of generality of k. □

Theorem 4.2. *Joye's regular 2-ary RL algorithm satisfies the* Generality of k *and* Secure generality. *The projective coordinates with CA formulae are its* Executable coordinates. *The affine and Jacobian coordinates are not the* Executable coordinates *of it.*

Proof.

1 *Generality of k*: Joye's regular m-ary (including 2-ary) RL algorithm satisfies the *Generality of k* that is shown in Lemma 4.1.

2 *Secure generality*: Joye's regular 2-ary RL algorithm (Alg. 15) can compute scalar multiplications regularly without dummy statements. "Without dummy statements" means that all statements of the algorithm decide the outputs for all valid input scalars. Therefore, Alg. 15 is a secure scalar multiplication algorithm with *Secure generality*.

3 *Executable coordinates*: In Algorithm 15, $R[0]$ and $R[1]$ are initialized as $\mathcal{O}$ in the Step 1, which cannot be represented by affine coordinates. In the main loop, the computation of $\mathcal{O} + P$ is required in Step 3. It is obvious that $\mathcal{O} + P$, $P + P$, and $-P + P$ are computed when $k = 1, 2, 0$ in Step 6, respectively. Therefore, Projective coordinates with CA formulae are *Executable coordinates*, not affine or Jacobian coordinates.

□

The related works of LR ECSMs are evaluated in the same manner as that of RL ECSMs, which is shown in Theorems 4.3 and 4.4.

Theorem 4.3. *The Montgomery ladder satisfies the* Generality of k. *The projective coordinates with CA formulae are its* Executable coordinates. *The affine and Jacobian coordinates are not the* Executable coordinates *of the Montgomery ladder.*

Proof.

1 *Generality of k*: Evidently, Alg. 16 can compute correctly when $k \in \mathbb{Z}/N\mathbb{Z}$ and $k_{\ell-1} = 1$.

The Montgomery ladder can also compute kP correctly if $k \in \mathbb{Z}/N\mathbb{Z}$ with $k_{\ell-1} = 0$ because the initial values of $R[0]$ and $R[1]$ are never changed even when scanning several '0's from the left beginning. $R[0]$ and $R[1]$ are maintained at their initial values until the scanning of the first '1' bit. Then, Alg. 16 works as if $k_{\ell-1} = 1$. Therefore, the Montgomery ladder can operate for any input scalar $k \in \mathbb{Z}/N\mathbb{Z}$ with $k_{\ell-1} = 0$ or $k_{\ell-1} = 1$, where ℓ is the bit length of an input scalar k. Alg. 16 is of the *Generality of k*.

2 *Secure generality*: Alg. 16 being able to compute regularly means that the computations for every bit of an input scalar are the same. Therefore, it can resist SPA. However, it contains dummy statements. If an input scalar k ends with several '0's, Step 3 of Alg. 16 becomes a dummy computation in some cases. We can learn whether an input scalar k ends with several '0's via an SEA on Step 3. Therefore, the Montgomery ladder does not satisfy *Secure generality* and is not a secure scalar multiplication algorithm against SCA.

3 *Executable coordinates*: Alg. 16 must compute an exceptional addition of $\mathcal{O} + P$ for any input scalar k, which cannot be computed by affine or Jacobian coordinates. It must also compute an exceptional doubling of $2\mathcal{O}$ if $k_{\ell-1} = 0$, which can-not be computed by affine coordinates. Evidently, the projective coordinates with CA formulae are *Executable coordinates* of it.

□

Lemma 4.2. *Joye's regular m-ary LR algorithm, Alg 17, can compute kP correctly with any input scalar* $k \in \{0, 1, ..., m-1\}^{\ell}$ *and* $P \in E(\mathbb{F}_p)$.

Proof. Let $k = \sum_{i=0}^{\ell-1} k_i m^i$ be an m-ary representation of an input scalar with $k_i \in [0, m-1]$.

Case 1: The MSB of k is not zero, $k_{\ell-1} \in [1, m-1]$. Joye's regular LR m-ary algorithm, which computes kP correctly, is shown in [Joy09].

Case 2: The MSB of k is zero, $k_{\ell-1} = 0$ and $k \neq 0$. From the initialization of Alg. 17, we have $R[.]s$ and A initialized as in Table 4.1. When Alg. 17 reads $k_{\ell-2} = 0$, $A = mA + R[1] = -mP +$

Table 4.1: Initialization of Alg. 17

$R[1]$	$R[2]$	...	$R[m]$	A
$(m-1)P$	mp	...	$(2m-2)P$	$-P$

$(m-1)P = -P$ is computed. When it reads $k_{\ell-3} = 0$, $A = mA + R[1] = -mP + (m-1)P = -P$ is computed. Until it reads the first bit k_i, where $k_i \neq 0$, A is always updated to $-P$. The value saved in $A = -P$ is never changed when scanning a series of '0's in front of the first $k_i \neq 0$. When Alg. 17 reads the first bit $k_i \neq 0$, $A = mA + R[k_i + 1] = -mP + (m + k_i - 1)P = (k_i - 1)P$ is computed. Then, we show that the value of A is updated to the same value when scalar k begins with MSB $k_{\ell-1} \neq 0$. If the MSB of k' is not zero, that is $k'_{\ell-1} \in [1, m-1]$, Alg. 17 reads

$k'_{\ell-1}$ and initializes $A = (k'_{\ell-1} - 1)P$, which is the same as the value of A when Alg. 17 reads k_i if $k_i = k'_{\ell-1}$, where the MSB of k is zero. Therefore, Joye's regular LR m-ary algorithm can also compute kP correctly here.

Case 3: $k = 0$, A is always updated to $-P$, and $-P + P = O$ is computed in the final correction.

Therefore, Joye's regular LR m-ary algorithm, Alg. 17, is of the *Generality of k*. □

Theorem 4.4. *Joye's regular 2-ary LR algorithm satisfies the* Generality of k *and* Secure generality. *The projective coordinates with CA formulae are its* Executable coordinates. *The affine and Jacobian coordinates are not the* Executable coordinates *of this algorithm.*

Proof.

1 *Generality of k*: Joye's regular m-ary (including 2-ary) LR algorithm satisfies the *Generality of k*, implying that it can compute kP for any input $k \in \mathbb{Z}/N\mathbb{Z}$ having $k_{\ell-1} = 0$ or $k_{\ell-1} = 1$, which is shown in Lemma 4.2.

2 *Secure generality*: Joye's regular 2-ary LR algorithm (Alg. 18) can compute scalar multiplications regularly without dummy statements. "Without dummy statements" means that all statements of the algorithm decide the outputs together, despite the valid input scalars. Therefore, Alg. 18 can resist SPA and SEA and is a secure scalar multiplication algorithm with *Secure generality*.

3 *Executable coordinates*: Alg. 18 contains exceptional inputs k. In Alg. 18, $R[1]$ and $R[2]$ are initialized as P and $2P$ in Step 1, respectively. A is initialized as $-P$ or $\mathcal{O}$ in Step 2 when $k_{\ell-1}$ is '0' or '1.' This means that Alg. 18 requires coordinates that can explicitly represent $\mathcal{O}$. We further investigate the main loop. The following cases of k are exceptional inputs: 1) either $2\mathcal{O} + P$ or $2\mathcal{O} + 2P$ appears in Step 4 if $k_{\ell-1}$ is '1' and 2) $-2P + 2P$ appears in Step 4 if $k_{\ell-1}$ is '0' and $k_{\ell-2}$ is '1.' Subsequently, we investigate the final correction. Evidently $\mathcal{O} + P$ appears if $k = 1$, $P + P$ appears if $k = 2$, and $-P + P$ appears if $k = 0$. In summary, Alg. 18 must compute addition or doubling with $\mathcal{O}$ if $k_{\ell-1}$ is '1' or $k = 1$; compute $P + P$ if $k = 2$; compute $P - P$ if $k_{\ell-1}$ is '0' and $k_{\ell-2}$ is '1,' or $k = 0$. Coordinates with CA formulae are *Executable coordinates*, not affine or Jacobian coordinates.

□

We herein focus on Algorithm 15 and 18, as they satisfy the secure generality.

4.3 Extended Elliptic Curve Affine Addition Formulae

Affine elliptic curve addition formulae are advantageous of using less memory. Their computational efficiency depends on the ratio of modular inversion cost to modular multiplication cost.

The detailed algorithms for affine elliptic curve addition formulae are shown in Algorithms 19–20. Both Algorithms 19 and 20 retain the value of the input point of P, which can be used continually as the next input. Affine elliptic curve addition formulae have exceptional computations. $\mathcal{O}$ cannot be represented explicitly in affine coordinates, while it is described as a point at infinity. Thus, affine elliptic curve addition formulae cannot compute $\mathcal{O}+P=\mathcal{O}$, $P-P=\mathcal{O}$, and $2P=\mathcal{O}$. The addition formula cannot compute $P+P$. The implementation of affine elliptic curve addition formulae requires conditional statements to process these exceptional computations.

Algorithm 19 Affine addition formula

Input: $P=(x_1,y_1)$ and $Q=(x_2,y_2)$
Output: $P+Q$

1: $t_0 \leftarrow (x_2-x_1)^{-1}$
2: $y_2 \leftarrow y_2-y_1$
3: $t_0 \leftarrow t_0 y_2$
4: $y_2 \leftarrow t_0^2-x_1-x_2$
5: $x_2 \leftarrow (x_1-y_2)t_0-y_1$
6: **return** (y_2,x_2)

Algorithm 20 Affine doubling formula

Input: $P=(x_1,y_1)$
Output: $2P$

1: $t_0 \leftarrow 3x_1^2+a$
2: $t_1 \leftarrow (2y_1)^{-1}$
3: $t_0 \leftarrow t_0 t_1$
4: $t_1 \leftarrow t_0^2-2x_1$
5: $t_2 \leftarrow (x_1-t_1)t_0-y_1$
6: **return** (t_1,t_2)

Therefore, we extend affine elliptic curve addition formulae to enable them to compute exceptional computations of $P-P=\mathcal{O}$ and $2P=\mathcal{O}$, which are shown in Equations (4.1)–(4.2). The detailed algorithms are shown in Algorithms 21–22, which can compute $P-P=\mathcal{O}$ and $2P=\mathcal{O}$ when $E(\mathbb{F}_p)$ does not include a point $(0,0)$. For example, $E(\mathbb{F}_p)$ without two-torsion points satisfies the condition, including the prime order elliptic curve on the Weierstrass form. Both Algorithms 21–22 retain the value of the input point of P similar to Algorithms 19–20. The main idea of extended affine elliptic curve addition formulae is that the modular inversion of zero that mathematically does not exist can be computed by the extended Euclidean algorithm, or Fermat's little theorem as zero. Therefore, let Algorithms 21–22 compute $(x_2-x_1)^{-1}$ and $(2y_1)^{-1}$ in the beginning by extended Euclidean algorithm or Fermat's little theorem and execute the remaining parts. The results for ordinary inputs P and Q are the same as those of Algorithms 19–20. The results for the exceptional computations of $P-P$ and $2P=\mathcal{O}$ are $(0,0)$ in Algorithms 21–22, which is assumed as $\mathcal{O}=(0,0)$.

$$\begin{aligned} x_3 &= [\frac{(y_2 - y_1)^2}{(x_2 - x_1)} - (x_1 + x_2)(x_2 - x_1)](x_2 - x_1)^{-1} \\ y_3 &= [(y_2 - y_1)(x_1 - x_3) - y_1(x_2 - x_1)](x_2 - x_1)^{-1} \end{aligned} \tag{4.1}$$

$$\begin{aligned} x_3 &= [(3x_1^2 + a)^2 - 8x_1y_1^2](4y_1^2)^{-1} \\ y_3 &= [(3x_1^2 + a)(x_1 - x_3) - 2y_1^2](2y_1)^{-1} \end{aligned} \tag{4.2}$$

The extended affine addition formula is derived from the traditional affine addition formula by factoring out $(x_2 - x_1)^{-1}$. The computational cost for Algorithm 21 is $6M + S + I$, and it requires the storage of seven field elements in memory. Similarly, the extended affine doubling formula is obtained from the traditional affine doubling formula by factoring out $(2y_1)^{-1}$. The computational cost of Algorithm 22 is $4M + 4S + I$, and it also requires the storage of seven field elements in memory.

Algorithm 21 Extended affine addition

Input: $P = (x_1, y_1)$ and $Q = (x_2, y_2)$
Output: $P + Q$

1: $t_0 \leftarrow (x_2 - x_1)^{-1}$
2: $y_2 \leftarrow y_2 - y_1$
3: $x_2 \leftarrow x_2 - x_1$
4: $t_1 \leftarrow (x_2 + 2x_1)x_2$
5: $x_2 \leftarrow y_1x_2$
6: $t_2 \leftarrow (y_2^2t_0 - t_1)t_0$
7: $t_1 \leftarrow ((x_1 - t_2)y_2 - x_2)t_0$
8: **return** (t_2, t_1)

Algorithm 22 Extended affine doubling

Input: $P = (x_1, y_1)$
Output: $2P$

1: $t_0 \leftarrow 3x_1^2 + a$, $t_1 \leftarrow (2y_1)^{-1}$
2: $t_4 \leftarrow y_1^2$, $t_2 \leftarrow 8x_1t_4$
3: $t_3 \leftarrow t_0^2 - t_2$, $t_2 \leftarrow t_1^2$
4: $t_3 \leftarrow t_3t_2$, $x_1 \leftarrow x_1 - t_3$
5: $t_0 \leftarrow t_0x_1$, $t_4 \leftarrow 2t_4$
6: $t_0 \leftarrow (t_0 - t_4)t_1$
7: $x_1 \leftarrow x_1 + t_3$
8: **return** (t_3, t_0)

Theorem 4.5. *Let $E(\mathbb{F}_p)$ be $y^2 = x^3 + ax + b$, $b \neq 0 \pmod p$ that excludes the point $(0, 0)$. For $P, Q \in E(\mathbb{F}_p)$, by setting $(0, 0)$ as $\mathcal{O}$, the extended addition formula can compute the addition of P and Q correctly if $P \neq Q$ ($P \neq \mathcal{O}$ and $Q \neq \mathcal{O}$), $P + Q = \mathcal{O}$, and $P = Q = \mathcal{O}$. The extended doubling formula can compute the doubling of P correctly for any point on $E(\mathbb{F}_p)$.*

Proof. Firstly we show that Algorithm 21 can compute $P + Q = \mathcal{O}$ and $\mathcal{O} + \mathcal{O} = \mathcal{O}$ correctly. For $P + Q = \mathcal{O}$ ($P = (x_1, y_1) = (x, y)$ and $Q = (x_2, y_2) = (x, -y)$), the inversion of zero ($(x_2 - x_1) = (x - x) = 0$) has to be computed. As we stated, by the extended Euclidean algorithm, or by Fermat's little theorem, the inversion of zero is computed to be zero. This demonstrates that by our Algorithm 21, $P + Q = \mathcal{O}$ is computed as:

$$x_3 = 0, y_3 = 0. \tag{4.3}$$

This implies $P + Q = (0, 0)$. We regard $(0, 0)$ as $\mathcal{O}$. Subsequently, the extended affine addition formula computes $P + Q = \mathcal{O}$ correctly. Further, it is clear that $\mathcal{O} + \mathcal{O} = \mathcal{O}$ ($\mathcal{O} = (0, 0)$) can also be computed correctly. We must emphasize that extracting the factor of $(x_2 - x_1)^{-1}$ does not affect the addition of other points, because the factor of $(x_2 - x_1)^{-1}$ becomes zero only when computing $P + Q = \mathcal{O}$ and $\mathcal{O} + \mathcal{O} = \mathcal{O}$. In the other situation, extracting the factor of $(x_2 - x_1)^{-1}$ is always safe.

Secondly we prove that Algorithm 22 can compute $2P = \mathcal{O}$ correctly. For $2P = \mathcal{O}$, where $P = (x_1, y_1) = (x_1, 0)$ or $P = \mathcal{O} = (0, 0)$ is of zero y-coordinate, the inversion of zero $2y_1 = 0$ has to be computed. Subsequently, $2P = (0, 0)$ is computed by extended affine doubling formula. We regard $(0, 0)$ as $\mathcal{O}$, implying that the extended affine doubling formula can compute $2P = \mathcal{O}$ correctly.

Further, extracting the factor of $(2y_1)^{-1}$ does not affect the doubling of other points. The y-coordinate of P becomes zero only when $2P = \mathcal{O}$.

The extended affine doubling formula is exception-free, implying that it can compute the doubling of all points on $E(\mathbb{F}_p)$ that excludes the point $(0, 0)$. □

It is important to note the traditional affine addition formulae are unable to correctly compute $P + Q = \mathcal{O}$, $P + \mathcal{O}$, and $2P = \mathcal{O}$, while our extended affine addition formulae can handle these exceptional cases of $P + Q = \mathcal{O}$ and $2P = \mathcal{O}$ correctly. Additionally, the Jacobian and projective addition formulae, which can also correctly compute $P + Q = \mathcal{O}$ and $2P = \mathcal{O}$, require the same exceptional computations as our extended affine addition formulae. This means that if our algorithms perform well using extended affine coordinates for ECSMs, they can be easily adapted to work with the Jacobian or projective addition formulae.

4.4 RL ECSMs

We improve Joye's regular 2-ary RL algorithm (Algorithm 15) to propose our scalar multiplication algorithms (Algorithms 23 and 24) that avoid exceptional computations such as $P + P$, $P + \mathcal{O}$, $P + Q = \mathcal{O}$ for additions, and $2P = \mathcal{O}$ for doubles. Then, we combine Algorithms 23 and 24 with (extended) affine addition formulae to secure RL ECSMs.

We enhance the efficiency of our Algorithm 23 and propose Algorithm 24 by two-bit scanning and using the affine double and quadruple formulae [LN12], which can compute both $2P$ and $4P$ simultaneously with only one modular inversion, denoted by $\{2P, 4P\} \leftarrow DQ(P)$. The

Algorithm 23 New 2-ary RL algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}], k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i, k_\ell \in \{0,1\}$
Output: kP
Uses: A, R[0], R[1]
Initialization
1: $R[0] = -P$
2: $R[1] = P$
3: $A \leftarrow 2P$
4: $R[k_0] \leftarrow R[k_0] + A$
Main loop
5: **for** $i = 1$ to $\ell - 1$ **do**
6: $\quad R[k_i] \leftarrow R[k_i] + A$
7: $\quad A \leftarrow 2A$
8: **end for**
Final correction
9: $R[k_0] \leftarrow R[k_0] - P$
10: $A \leftarrow -A + R[0] + 2R[1]$
11: $A = (-1)^{k_\ell} \times A$
12: **return** A

computational cost of affine double and quadruple formulae is $8M + 8S + I$, where M, S, and I are the computational cost of modular multiplication, modular squaring, and modular inversion, respectively.

Both Algorithms 23 and 24 make the assumption that k belongs to the set $\mathbb{Z}/N\mathbb{Z}$ and is in the range $[-\frac{N}{2}, \frac{N}{2}]$. This is a natural setting as k is determined by the modulo N. This technique ensures that our algorithms exactly exclude exceptional computations, as proven in Theorem 4.6. Then, the scalar k is represented by $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$ $(k_i \in \{0,1\})$, where k_ℓ is the sign bit and $0 \leq |k| \leq \frac{N}{2}$. In Algorithm 24, we adjust the bit length of $|k|$ to be odd by padding a "0" in front of the input scalar $|k|$. This allows for two-bit scanning to be performed effectively by Alg. 24.

Algorithms 23 and 24 are divided into three parts: initialization, main loop, and final correction. In comparison to Algorithm 15, the initialization of $R[.]$ is modified to eliminate the exceptional initialization of $\mathcal{O}$ and the exceptional computation $\mathcal{O} + P$ in the main loop. This initialization of $R[.]$ results in $R[0] + 2R[1] = P$ being added to the final result, thus avoiding the exceptional computation in the original final correction of Algorithm 15, $A \leftarrow A + P$. Step 3 of Algorithms 23 and 24 eliminates the exceptional computations of $P + P$ or $P - P$ if A is initialized as P. The final correction adjusts the excess computations made by Step 3 in Algorithms 23 and 24.

In Algorithms 23 and 24, the (extended) affine addition formulae (traditional and our extended version) are used in different steps. The traditional affine addition formulae are used in Steps 1–9 of Algorithm 23 and Steps 1–11 of Algorithm 24. The extended affine addition formulae are used only once in the final addition of Step 10 of Algorithm 23 and the final addition of Step 12 of Algorithm 24. Actually, the use of extended affine addition formulae is only

Algorithm 24 New two-bit 2-ary RL algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $k_\ell \in \{0, 1\}$
Output: kP
Uses: A, A[1], R[0], R[1]
Initialization
1: $R[0] = -P$
2: $R[1] = P$
3: $\{A, A[1]\} \leftarrow DQ(P) = \{2P, 4P\}$
4: $R[k_0] \leftarrow R[k_0] + A$
Main loop
5: **for** $i = 1$ to $\ell - 1$ **do**
6: $\quad R[k_i] \leftarrow R[k_i] + A$
7: $\quad R[k_{i+1}] \leftarrow R[k_{i+1}] + A[1]$
8: $\quad \{A, A[1]\} \leftarrow DQ(A[1])$
9: $\quad i = i + 2$
10: **end for**
Final correction
11: $R[k_0] \leftarrow R[k_0] - P$
12: $A \leftarrow -A + R[0] + 2R[1]$
13: $A = (-1)^{k_\ell} \times A$
14: **return** A

necessary when the input scalar $k = 0$. If $k = 0$ is guaranteed to be excluded from the input, then only the traditional affine addition formulae are needed in the whole Algorithms 23 and 24. Our Algorithms 23 and 24 satisfy secure generality. Theorem 4.6 proves that Algorithms 23–24 avoid all exceptional computations of (extended) affine addition formulae when $k \in [-\frac{N}{2}, \frac{N}{2}]$, $N \neq 3$.

Theorem 4.6. *Let $E(\mathbb{F}_p)$ be an elliptic curve without two-torsion points. Let $P \in E(\mathbb{F}_p)$, $P \neq \mathcal{O}$ be an elliptic curve point, whose order is $N \neq 3$. Then, Algorithms 23 and 24 using (extended) affine addition formulae can compute kP correctly for any input $k \in [-\frac{N}{2}, \frac{N}{2}]$.*

Proof. We prove that all three parts exclude the exceptional computations of (extended) affine, which are additions of $P \pm P$ and $\mathcal{O} + P$, and doubling of $2P = \mathcal{O}$. The doubling of $2P = \mathcal{O}$ does not appear in the algorithms because of the assumption that $E(\mathbb{F}_p)$ is without two-torsion points. Thus, we only focus on exceptional additions.

In the initialization, $R[0]$ and $R[1]$ initialized as $(P_x, -P_y)$ and (P_x, P_y) are "odd" scalar points such as $(2t+1)P, t \in \mathbb{Z}$. A initialized as $((2P)_x, (2P)_y)$ is an "even" scalar point such as $(2t)P, t \in \mathbb{Z}$. It is obvious that $R[0] \leftarrow -P + 2P$ or $R[1] \leftarrow P + 2P$ in Step 4 can be computed correctly by the original affine formula if $N \neq 3$.

In the main loop, it is noteworthy that 1) $A \neq \mathcal{O}$ because of $E(\mathbb{F}_p)$ without two-torsion points and A always increases as an "even" scalar point until $2^\ell P, 2^\ell < N$ because of $|k| \leq \frac{N}{2}$. 2) $R[0] \neq \mathcal{O}$ is always updated as an "odd" scalar point with a smaller scalar than A after a loop computation. 3) $R[1] \neq \mathcal{O}$ is also always updated as an "odd" scalar point. If $|k| = \{1\}^\ell$, $R[1]$ is always with a larger scalar than A and becomes to $(2^\ell + 1)P, (2^\ell + 1) \leq N$ at the end of loop.

If $N = (2^\ell + 1)$ then $|k| = \{1\}^\ell$ can not be a valid input which is larger than $\frac{N}{2}$. Thus, $R[1]$ can be with a larger scalar than A but never equals to $NP = \mathcal{O}$. In summary, $R[0], R[1], A \neq \mathcal{O}$ are scalar points of P whose scalars are never over N. Therefore, during the main loop, exceptional computations $\mathcal{O} + P$ and $P + Q = \mathcal{O}$ does not appear and exceptional addition $P + P$ does not appear because that the "odd" scalar point ($R[.]$) can never be the same point as the "even" scalar point (A). The computations in the main loop exclude the exceptional computations of affine.

In the final correction, exceptional computation $\mathcal{O} + P$ dose not appear in Step 9. After the main loop, $R[k_0] \neq \mathcal{O}$ is shown. Exceptional computation $P + P$ dose not appear in Step 9. $R[k_0]$ is an "odd" scalar point and $-P = (N-1)P$ is an "even" scalar point. They can not be the same. Step 9 computes $P - P = \mathcal{O}$ when only $k_0 = 0$. However, we can always put "0" in front of k to avoid this if $k_{\ell-1} \neq 0$. This additional "0" causes $R[0] \leftarrow R[0] + 2^\ell P$ which can also be computed correctly in Step 6 and A is updated as $(2^{\ell+1})P$. In Step 10 of Algorithm 23 and Step 12 of Algorithm 24, $-A$, $R[0]$, $2R[1]$ can not be $\mathcal{O}$ which excludes exceptional computation of $P + \mathcal{O}$. $R[0]$ has smaller scalar than A which excludes exceptional computation of $P - P$. If $-A = R[0]$ which means that the input scalar k is even and $k_0 = 0$, $-A = (N - 2^\ell)$ is with "odd" scalar and $R[0] \leftarrow R[0] - P$ computed in Step 9 or 11 is with "even" scalar. They can not be the same where we have a contradiction. So $-A \neq R[0]$ excludes exceptional computation $P + P$. If $k = 0$, the last addition of Step 10 in Algorithm 23 and Step 12 in Algorithm 24 compute the exceptional computation, $P - P$. Our extended affine formula can be used here because of $E(\mathbb{F}_p)$ without two-torsion points excludes point $(0, 0)$ to avoid this.

The two-bit scanning version analysis proceeds in an analogous way. □

Chapter 5

SCAs Resistant and Compact LR ECSMs

In this chapter, we focus on LR ECSMs, which are expected to use less memory than RL ECSMs. To enhance the computational efficiency of ECSMs using affine addition formulae, we first consider 8 affine combination addition formulae, which enhance the efficiency by reducing the number of modular inversions in Section 5.1. Based on the three notions of SCAs resistant ECSMs in Chapter 4, we improve Joye's regular 2-ary LR algorithm to make sure (extended) affine addition formulae can be used without introducing SCA weaknesses and propose our SCAs resistant and compact LR ECSMs in Section 5.2. We also revisit proposed RL ECSMs in Chapter 4 using affine combination addition formulae to improve their efficiency.

5.1 Affine Combination Addition Formulae

Any two or more inversions can be computed using only a single inversion by applying the Montgomery trick, computing x_1^{-1} and x_2^{-1} as $x_2 I$, and $x_1 I$, where $I = (x_1 x_2)^{-1}$. The computational cost of nI becomes $3(n-1)M + I$, which is more efficient when $I > 3M$. Montgomery trick can be easily applied to independent inversion computations, but it is not directly applicable to dependent inversion computations. By combining affine elliptic curve additions and using the Montgomery trick, various affine combination-addition formulae are refined and proposed.

First, we refine an affine doubling and addition algorithm (DA algorithm), computing $2P+Q$ as $P+Q+P$ with $P(x_1, y_1)$, $Q(x_2, y_2)$ [ELM03], and an affine double and quadruple algorithm (DQ algorithm), computing $2P$ and $4P$ together [LN12]. Neither [ELM03] nor [LN12] makes the amount of memory required in their algorithms clear. We improve the DA and DQ algorithms to optimize their memory usage as seven field elements in Algs. 25 and 27, respectively.

The DA formulae proposed by Eisentrager et al. are shown as [ELM03]:

$$\begin{aligned} x_3 &= \lambda_1^2 - x_1 - x_2, \quad y_3 = \lambda_1(x_1 - x_3) - y_1 \\ x_4 &= (\lambda_2 - \lambda_1)(\lambda_2 + \lambda_1) + x_2, \quad y_4 = \lambda_2(x_1 - x_4) - y_1 \\ \lambda_1 &= \frac{y_2 - y_1}{x_2 - x_1}, \quad \lambda_2 = -\lambda_1 - \frac{2y_1}{x_3 - x_1} \end{aligned} \tag{5.1}$$

The inversion of $(x_2 - x_1)(x_3 - x_1)$ must be computed first when using the Montgomery trick. However, x_3 depends on the inversion computation of $(x_2 - x_1)$. Therefore, the DA algorithm computes the inversion of $(x_2 - x_1)^3(x_3 - x_1) = (x_2 - x_1)(y_2 - y_1)^2 - (2x_1 + x_2)(x_2 - x_1)^3$ first, and then computes the inversions of $(x_2 - x_1)$ and $(x_3 - x_1)$. Two tricks are used to reduce the computational cost of the DA algorithm: λ_2 can be computed by λ_1, and the result (x_4, y_4) can be computed without fully computing (x_3, y_3). The computational cost is $9M + 2S + I + 15A$, and the memory cost is revised to seven field elements (Alg. 25).

The exceptional inputs for the DA algorithm are: $P = \mathcal{O}$, $Q = \mathcal{O}$, $2P + Q = \mathcal{O}$, $P + Q = \mathcal{O}$, and $P = Q$. P and Q have the same x-coordinate when $P+Q = \mathcal{O}$ and $P = Q$. If an exceptional input $P + Q = \mathcal{O}$ or $P = Q$ is computed using Alg. 25, $-P$ is its output. This means that $DA(P, Q) \to -P$ for $P + Q = \mathcal{O}$ or $P = Q$. If Alg. 25 is used twice with the exceptional input $P + Q = \mathcal{O}$ (e.g., $P = (x, y)$ and $Q = -P = (x, -y)$), then Alg. 25 outputs $DA(P, -P) \to -P$ first. Next, if we input the updated P and original Q into the algorithm again, then Alg. 25 outputs $DA(-P, -P) \to -(-P)$. Fortunately, this is the correct result of $2P + Q = P$ for $P + Q = \mathcal{O}$. In summary, using Alg. 25 twice results in $DA(DA(P, Q), Q) = P$ $(P + Q = \mathcal{O})$, which is the correct result of $2P + Q$ $(P + Q = \mathcal{O})$.

Algorithm 25 DA algorithm
Memory: 4+3=7 field elements.
Computational cost: $9M + 2S + I + 15A$.

Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$
Output: $P = 2P + Q$

1: $y_2 = y_2 - y_1$, $t_0 = y_2^2$, $t_1 = 2x_1 + x_2$
2: $x_2 = x_2 - x_1$, $t_2 = x_2^2$, $t_1 = t_1 t_2$
3: $t_0 = t_0 - t_1$, $t_1 = (t_0 x_2)^{-1}$
4: $t_0 = t_0 t_1 y_2$, $t_1 = -2t_1 x_2 t_2 y_1 - t_0$, $t_2 = t_1 + t_0$
5: $t_0 = (t_1 - t_0)t_2 + x_2 + x_1$, $x_2 = x_2 + x_1$
6: $x_1 = (x_1 - t_0)t_1 - y_1$, $y_2 = y_2 + y_1$, $y_1 = x_1$, $x_1 = t_0$
7: **return** (x_1, y_1)

Alg. 25 cannot compute $2P + Q$ correctly when $P + Q = \mathcal{O}$ or $P = Q$, but the computation of $2P + Q$ where $P + Q = \mathcal{O}$ is necessary in Alg. 34. Therefore, we refine DA algorithm to a new DA algorithm (Alg. 26) such that it can compute $2P + Q$ correctly when $P + Q = \mathcal{O}$ or $P = Q$. The idea is computing $2P$ followed by $2P + Q$ instead of computing $P + Q$ and then $(P + Q) + P$. In this case, x_3 depends on the inversion of $2y_1$, where x_3 is the x-coordinate of $2P$. Thus, applying the Montgomery trick is not straightforward. Therefore, the inversion of $(2y_1)^3(x_3 - x_2) = 2y_1\left[(3x_1^2 + a)^2 - 4y_1^2(2x_1 + x_2)\right]$ is computed, which induces the inversions of

both $2y_1$ and $x_3 - x_2$. Alg. 26 correctly computes $2P + Q$ when $2P \neq \mathcal{O}$, $2P \neq Q$, $2P + Q \neq \mathcal{O}$, and $Q \neq \mathcal{O}$.

Algorithm 26 New DA algorithm
Memory: 4+4=8 field elements.
Computational cost: $9M + 5S + I + 17A$.

Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$
Output: $P = 2P + Q$
1: $t_0 = (2y_1)^2$, $x_2 = x_2 + 2x_1$, $t_1 = -t_0 x_2$
2: $x_2 = x_2 - 2x_1$, $t_2 = 3x_1^2 + a$, $t_3 = t_2^2$
3: $t_1 = t_1 + t_3$, $t_3 = (2t_1 y_1)^{-1}$
4: $t_2 = t_2 t_3 t_1$, $t_3 = 2t_3 y_1 t_0$, $t_1 = t_2^2 - 2x_1$
5: $t_0 = ((x_1 - t_1)t_2 - y_1 - y_2)t_3$
6: $x_1 = t_0^2 - x_2 - t_1$, $y_1 = (x_2 - x_1)t_0 - y_2$
7: **return** (x_1, y_1)

Next, the DQ algorithm [LN12] is refined. The computational cost is $8M + 8S + I + 18A$. The memory usage is improved to seven field elements (Alg. 27). If we require only the result of $4P$, the memory usage of the DQ algorithm can be further improved to six field elements because the memory to save $2P$ can be reduced.

Algorithm 27 DQ algorithm
Memory: 4+3=7 field elements.
Computational cost: $8M + 8S + I + 18A$.

Input: $P(x_1, y_1)$, $Q(x_2, y_2)$
Output: $Q = 2P$, $P = 4P$
1: $t_0 = x_1^2$, $x_2 = 2y_1^2$, $t_1 = x_2^2$
2: $x_2 = 3((x_2 + x_1)^2 - t_0 - t_1)$, $t_0 = 3t_0 + a$, $t_2 = t_0^2$
3: $x_2 = (x_2 - t_2)t_0$, $t_1 = 2t_1$, $x_2 = x_2 - t_1$
4: $t_2 = 2x_2 y_1$, $t_2 = t_2^{-1}$, $t_0 = t_0 x_2 t_2$
5: $x_2 = t_0^2 - 2x_1$, $y_2 = (x_1 - x_2)t_0 - y_1$, $t_2 = t_1 t_2$
6: $t_0 = (3x_2^2 + a)t_2$, $x_1 = t_0^2 - 2x_2$, $y_1 = (x_2 - x_1)t_0 - y_2$
7: **return** (x_2, y_2), (x_1, y_1)

Alg. 28 combines two continuous affine additions. The inversion of $(x_2 - x_1)^3(x_4 - x_3) = (x_2 - x_1)(y_2 - y_1)^2 - (x_1 + x_2 + x_3)(x_2 - x_1)^3$ induces two inversions of $(x_2 - x_1)$ and $(x_4 - x_3)$, where x_4 is the x-coordinate of $P + Q$. Alg. 28 computes $P + Q + R$ correctly if $P + Q \neq \mathcal{O}$, $P + Q + R \neq \mathcal{O}$, $P \neq Q$, $P + Q \neq R$, and $P, Q, R \neq \mathcal{O}$.

We now consider $4P + Q$. The computations $P \leftarrow 2P$, $P \leftarrow 2P$, and $P \leftarrow P + Q$, each of which costs I, are required to obtain $4P + Q$. A new affine quadruple and addition algorithm (Alg. 29) that combines all these affine additions are proposed, which requires an inversion instead of three inversions. The inversion of $(x_4 - x_2)(2y_3)^2(2y_1)^8$ induces the inversions of $(x_4 - x_2)$, $2y_3$, and $2y_1$. Alg. 29 computes $4P + Q$ correctly if $4P + Q \neq \mathcal{O}$, $4P \neq Q$, and $P, 2P, 4P, Q \neq \mathcal{O}$.

Algs. 30 and 31 compute $(P = P + Q,\ Q = 2Q)$ and $(P = P + Q,\ R = R + T)$, respectively,

Algorithm 28 Two-continuous-adds
Memory: 6+4=10 field elements.
Computational cost: $9M + 4S + I + 14A$.

Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $R = (x_3, y_3)$
Output: $Q = P + Q + R$

1: $y_2 = y_2 - y_1$, $t_0 = y_2^2$, $t_1 = x_2 - x_1$
2: $t_2 = t_1^2$, $t_3 = t_0 - (x_1 + x_2 + x_3)t_2$
3: $t_0 = (t_1 t_3)^{-1}$, $y_2 = y_2 t_0 t_3$
4: $t_3 = y_2^2 - x_1 - x_2$, $x_2 = (x_1 - t_3)y_2 - y_1 - y_3$
5: $t_0 = t_0 t_1 t_2 x_2$, $x_2 = t_0^2 - x_3 - t_3$
6: $y_2 = (x_3 - x_2)t_0 - y_3$
7: **return** (x_2, y_2)

Algorithm 29 Quadruple-add
Memory: 4+6=10 field elements.
Computational cost: $18M + ma + 14S + I + 40A$.

Input: $P(x_1, y_1)$, $Q(x_2, y_2)$
Output: $P = 4P + Q$

1: $t_0 = x_1^2$, $t_1 = 2y_1^2$, $t_2 = t_1^2$, $t_3 = (t_1 + x_1)^2 - t_0 - t_2$
2: $t_0 = 3t_0 + a$, $t_4 = t_0^2 - 2t_3$, $t_3 = t_0(t_3 - t_4) - 2t_2$
3: $t_1 = 2t_1 x_2 + 2t_4$, $t_5 = t_3^2$, $t_1 = 4t_1 t_5$, $t_5 = 4t_2 a$
4: $t_4 = (3t_4^2 + t_5)^2 - t_1$, $t_5 = (2t_3 t_4 y_1)^{-1}$
5: $t_1 = 2t_2 t_4 t_5$, $t_5 = t_3 t_5$, $t_0 = t_0 t_4 t_5$, $t_2 = t_2^2$, $t_5 = 32 t_2 t_5 y_1$
6: $t_2 = t_0^2 - 2x_1$, $x_1 = (x_1 - t_2)t_0 - y_1$, $y_1 = 3t_2^2 + a$, $t_1 = t_1 y_1$
7: $y_1 = t_1^2 - 2t_2$, $t_2 = (t_2 - y_1)t_1 - x_1$, $x_1 = 4x_1^2$, $t_5 = t_5 x_1$
8: $t_2 = (t_2 - y_2)t_5$, $x_1 = t_2^2 - x_2 - y_1$, $y_1 = (x_2 - x_1)t_2 - y_2$
9: **return** (x_1, y_1)

using only one inversion. We can also combine affine double and quadruple computations with two independent affine additions ($P = P + Q$, $R = R + T$, $Q = 2T$, $T = 4T$) into Alg. 32.

Table 5.1 summarizes the affine combination-addition formulae used in the ECSMs to reduce the number of inversions.

Table 5.1: Affine combination-addition algorithms

Affine combination-addition algorithms	Ordinary	Applying the Montgomery trick	memory
DA-algorithm Alg. 25 ($P = 2P + Q$) [ELM03]	$4M + 3S + 2I + 12A$	$9M + 2S + I + 15A$	7
DQ-algorithm Alg. 27 ($Q = 2P, P = 4P$) [LN12]	$4M + 4S + 2I + 12A$	$8M + 8S + I + 18A$	7
New DA algorithm Alg. 26 ($P = 2P + Q$)	$4M + 3S + 2I + 12A$	$9M + 5S + I + 17A$	8
Two-continuous-adds Alg. 28 ($Q = P + Q + R$)	$4M + 2S + 2I + 12A$	$9M + 4S + I + 14A$	10
Quadruple-add Alg. 29 ($P = 4P + Q$)	$6M + 5S + 3I + 18A$	$18M + ma + 14S + I + 40A$	10
Independent add and double Alg. 30 ($P = P + Q$, $Q = 2Q$)	$4M + 3S + 2I + 12A$	$7M + 3S + I + 15A$	7
Independent two adds Alg. 31 ($P = P + Q$, $R = R + T$)	$4M + 2S + 2I + 12A$	$7M + 2S + I + 14A$	10
Independent adds and DQ Alg. 32 ($P = P + Q$, $R = R + T$, $Q = 2T$, $T = 4T$)	$8M + 6S + 4I + 24A$	$21M + 10S + I + 34A$	12

5.2 LR ECSMs

We revisit the two-bit 2-ary RL algorithm (Alg. 23) by utilizing Algs. 31–32 to decrease the number of inversions in a loop. We propose a new extended exceptional-free 4-ary RL algorithm

Algorithm 30 Independent add and double
Memory: 4+3=7 field elements.
Computational cost: $7M + 3S + I + 15A$.

Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$
Output: $P = P + Q$, $Q = 2Q$
1: $t_0 = (2(x_2 - x_1)y_2)^{-1}$, $t_1 = 2(y_2 - y_1)t_0 y_2$
2: $t_2 = t_1^2 - x_1 - x_2$, $x_1 = x_2 - x_1$, $t_0 = t_0 x_1$
3: $x_1 = (x_2 - x_1 - t_2)t_1 - y_1$, $y_1 = x_1$, $x_1 = t_2$
4: $t_1 = (3x_2^2 + a)t_0$, $t_0 = t_1^2 - 2x_2$
5: $x_2 = (x_2 - t_0)t_1 - y_2$, $y_2 = x_2$, $x_2 = t_0$
6: **return** (x_1, y_1), (x_2, y_2)

Algorithm 31 Independent two adds
Memory: 8+2=10 field elements.
Computational cost: $7M + 2S + I + 14A$.

Input: $P(x_1, y_1)$, $Q(x_2, y_2)$, $R(x_3, y_3)$, $T(x_4, y_4)$
Output: $P = P + Q$, $R = R + T$
1: $x_1 = x_1 - x_2$, $x_3 = x_3 - x_4$, $t_0 = (x_1 x_3)^{-1}$
2: $y_1 = y_1 - y_2$, $t_1 = t_0 x_3 y_1$
3: $t_0 = t_0 x_1$, $x_1 = x_1 + x_2$, $y_1 = t_1^2 - x_1 - x_2$
4: $x_1 = y_1$, $y_1 = (x_2 - x_1)t_1 - y_2$, $y_3 = y_3 - y_4$
5: $t_0 = t_0 y_3$, $x_3 = x_3 + x_4$, $t_1 = t_0^2$
6: $x_3 = t_1 - x_3 - x_4$, $y_3 = (x_4 - x_3)t_0 - y_4$
7: **return** (x_1, y_1), (x_3, y_3)

(Alg. 33), called the EX-EF 4-ary RL algorithm, in which the two additions within the loop are independent of one another. The core idea is to update $R[0]$ (respectively $R[1]$) into $(R[0], R[2])$ (respectively $R[1]$ and $R[3]$). This ensures that the array $R[k_i]$ cannot be the same as $R[k_{i+1}+2]$ in Step 5, allowing Algs. 31 and 32 to be used in Alg. 33. Theorem 4.6 can also be extended to Alg. 33. The 2-ary RL algorithm in Chapter 4 can be combined with Alg. 30 to reduce the number of inversions to one in a loop.

We improve Alg. 18 into a new exception-free 2-ary LR algorithm (Alg. 34), called the EF 2-ary LR algorithm, and a new exception-free 4-ary LR algorithm (Alg. 35), called the EF 4-ary LR algorithm. Both of these algorithms avoid the exceptional computations of $P + P$, $P + Q = \mathcal{O}$, $P + \mathcal{O}$, and $2P = \mathcal{O}$. Alg. 35 enhances the efficiency of Alg. 34 by using a two-bit scanning with the DQ algorithm. The DQ algorithm is denoted by $\{2P, 4P\} \leftarrow DQ(P)$. The quad algorithm only outputs the quadruple point from the DQ algorithm, which is denoted by $4P \leftarrow Quad(P)$. We also combine Alg. 34 (or Alg. 35) with (extended) affine and affine combination-addition algorithms.

Algs. 34 and 35 assume that $k \in \mathbb{Z}/N\mathbb{Z}$ lies within the range of $k \in [-\frac{N}{2}, \frac{N}{2}]$. k is determined by taking the modulo of N, making this a natural setting. This technique ensures that our algorithms exclude exceptional computations, as proven by Theorem 5.1. k is then represented by $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $(k_i \in \{0, 1\})$, where $k_\ell \in \{0, 1\}$ is the sign bit, and ℓ is the number of bits of $|k|$, $l \leq \texttt{bitlen}(N) - 1$. Alg. 35 need to adjust the length of the input scalars, including

Algorithm 32 Independent adds and DQ
Memory: 8+4=12 field elements.
Computational cost: $21M + 10S + I + 34A$.

Input: $P(x_1, y_1)$, $Q(x_2, y_2)$, $R(x_3, y_3)$, $T(x_4, y_4)$
Output: $P = P + Q$, $R = R + T$, $Q = 2T$, $T = 4T$

1: $t_0 = x_4^2$, $t_1 = 2y_4^2$, $t_2 = t_1^2$
2: $t_1 = 3((t_1 + x_4)^2 - t_0 - t_2)$, $t_0 = 3t_0 + a$
3: $t_3 = t_0^2$, $t_1 = (t_1 - t_3)t_0$, $t_2 = 2t_2$
4: $t_1 = t_1 - t_2$, $x_2 = x_2 - x_1$, $x_3 = x_3 - x_4$
5: $t_3 = (2y_4x_2x_3t_1)^{-1}$, $y_2 = y_2 - y_1$
6: $y_2 = 2y_2y_4t_3x_3t_1$, $t_3 = t_3x_2$, $y_3 = y_3 - y_4$
7: $y_3 = 2y_3y_4t_3t_1$, $t_3 = t_3x_3$, $t_0 = t_0t_3t_1$
8: $t_2 = t_2t_3$, $t_1 = y_2^2 - x_2 - 2x_1$, $x_1 = (x_1 - t_1)y_2 - y_1$
9: $y_1 = x_1$, $x_1 = t_1$, $t_1 = y_3^2 - x_3 - 2x_4$
10: $x_3 = (x_4 - t_1)y_3 - y_4$, $y_3 = x_3$, $x_3 = t_1$
11: $x_2 = t_0^2 - 2x_4$, $y_2 = (x_4 - x_2)t_0 - y_4$
12: $t_1 = 3x_2^2 + a$, $t_1 = t_1t_2$
13: $x_4 = t_1^2 - 2x_2$, $y_4 = (x_2 - x_4)t_1 - y_2$
14: **return** (x_1, y_1), (x_3, y_3), (x_2, y_2), (x_4, y_4)

the sign bit, to be even by padding a '0' bit after the sign bit of the input scalars, thus allowing for two-bit scanning to operate well for both even and odd input scalar lengths.

Algs. 34 and 35 are composed of three parts: initialization, main loop, and final correction. Unlike Alg. 18, we modify the initialization of $R[.]$ and A to eliminate the exceptional initialization of $A \leftarrow \mathcal{O}$ when $k_{\ell-1} = 1$ and the exceptional computation of $2\mathcal{O} + P$, $2\mathcal{O} + 2P$, and $-2P + 2P$ in the main loop. Our new initialization of $R[.]$ and A adds $2P$ to the final result, which we then subtract in Step 5 of the final correction of Alg. 34 and Step 10 of the final correction of Alg. 35; thus, we avoid the exceptional computation, $A \leftarrow A + R[0]$, in the original final correction of Alg. 18.

Next, we explain how the extended affine and affine addition formulae are used in Algs. 34 and 35. The affine addition formulae are used in Steps 1–5 of Alg. 34 and Steps 1–10 of Alg. 35. On the other hand, the extended affine addition formulae are only used once in Step 6 of Alg. 34 and Step 11 of Alg. 35. The extended affine addition formulae are only necessary when the input scalar is equal to 0. If this scenario is avoided, the Algs. 34 and 35 can be performed using only the affine addition formulae.

To improve the efficiency of Alg. 34, we apply the two DA algorithms, Algs. 25 and 26, in Steps 3 and 5. While Alg. 25 cannot be used directly in Alg. 34 due to the exceptional computation of $A + R[k_i] = \mathcal{O}$ when $k_{\ell-1} = 0$. Using Alg. 25 twice with the exceptional inputs $A + R[k_i] = \mathcal{O}$ when $k_{\ell-1} = 0$ allows for the correct result of $A \leftarrow 2P$ to be computed. To ensure that Alg. 34 with Alg. 25 can compute correctly, we adjust the number of '0's from $k_{\ell-1}$ until the first '1' bit on the left is even by padding a '0' after the sign bit of k. On the other hand, Alg. 26 can be directly used without any adjustments. Additionally, we apply the two-continuous-adds algorithm (Alg. 28) in Step 7 of Alg. 35 to reduce the number of inversions to one for every bit

Algorithm 33 EX-EF 4-ary RL algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $k_\ell \in \{0, 1\}$
Output: kP
Uses: A_0, A_1, $R[0]$, $R[1]$, $R[2]$, $R[3]$
Initialization
1: $R[0] = -P$, $R[1] = P$, $\{A_0, A_1\} \leftarrow DQ(P)$
2: $R[k_0] \leftarrow R[k_0] + A_0$, $R[k_1] \leftarrow R[k_1] + A_0$
3: $R[k_2 + 2] \leftarrow A_1$, $R[\neg k_2 + 2] \leftarrow -A_0$, $\{A_0, A_1\} \leftarrow DQ(A_1)$
Main loop
4: **for** $i = 3$ to $\ell - 1$ **do**
5: $R[k_i] \leftarrow R[k_i] + A_0$, $R[k_{i+1} + 2] \leftarrow R[k_{i+1} + 2] + A_1$
6: $\{A_0, A_1\} \leftarrow DQ(A_1)$
7: $i = i + 2$
8: **end for**
Final correction
9: $R[0] = R[0] + R[2]$, $R[1] = R[1] + R[3]$
10: $R[\neg k_2] = R[\neg k_2] + 2P$, $R[k_0] \leftarrow R[k_0] - P$
11: $A_0 \leftarrow -A_0 + R[0] + 2R[1]$
12: $A_0 = (-1)^{k_\ell} \times A_0$
13: **return** A_0

computation.

Algs. 34 and 35 satisfy the *Secure generality*, and the affine coordinates are *Executable coordinates* for them. Notably, Algs. 34 and 35 do not contain any conditional or dummy statements, making them resistant to both SPA and SEA. The extended affine, affine, and affine combination-addition algorithms are employable without introducing conditional statements, as indicated in Theorem 5.1. Thus, Algs. 34 and 35 with (extended) affine combined with affine combination-addition algorithms are secure ECSM algorithms.

Theorem 5.1. *Let $E(\mathbb{F}_p)$ be an elliptic curve without two torsion points. Let $P \in E(\mathbb{F}_p)$ and $P \neq \mathcal{O}$ be an elliptic curve point whose order is $N > 4$. Then, Algs. 34 and 35 using (extended) affine and affine combination-addition formulae can compute kP correctly without exceptional computations for any input $k \in [-\frac{N}{2}, \frac{N}{2}]$.*

Proof. We prove that none of the three parts have exceptional computations of affine addition formulae, which are additions of $P \pm P$, $\mathcal{O} + P$ and doubling of $2P = \mathcal{O}$. The doubling of $2P = \mathcal{O}$ does not appear in the algorithms because of the assumption that $E(\mathbb{F}_p)$ does not have two torsion points. Thus, we focus only on exceptional additions.

In the initialization of Alg. 34, A, $R[0]$, and $R[1]$ are initialized as $((2P)_x, (2P)_y)$, $((2P)_x, -(2P)_y)$, and $(P_x, -P_y)$, respectively.

In the main loop of $A \leftarrow 2A + R[k_i]$, an exceptional addition of $\mathcal{O} + P$ does not appear. $2A$ cannot be $\mathcal{O}$ because $E(\mathbb{F}_p)$ does not have two torsion points, and $R[k_i] \neq \mathcal{O}$ is initialized as a fixed elliptic curve point.

An exceptional addition of $P + Q = \mathcal{O}$ does not appear. $A = k_A P$ is updated as a scalar point of P by $A \leftarrow 2A + R[k_i]$ in the main loop. Subsequently, the scalar of k_A is updated as

Algorithm 34 EF 2-ary LR algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $k_l \in \{0, 1\}$
Output: kP
Uses: A, $R[0]$, $R[1]$
Initialization
1: $A \leftarrow 2P$, $R[0] \leftarrow -2P$, $R[1] \leftarrow -P$
Main loop
2: **for** $i = \ell - 1$ to 1 **do**
3: $\quad A \leftarrow 2A + R[k_i]$
4: **end for**
Final correction
5: $A \leftarrow 2A + R[0]$
6: $A \leftarrow A + R[k_0]$
7: $A \leftarrow (-1)^{k_\ell} \times A$
8: **return** A

$k_A = 2k_A - 2$ or $k_A = 2k_A - 1$ every iteration. Thus, k_A is updated to $2 \leq k_A \leq 2^{\ell-1} + 1$ at the end of the main loop, where $1 \leq \ell \leq \texttt{bitlen}(N) - 1$ is the bit length of the input $|k| \in [0, \frac{N}{2}]$. Therefore, $2 \leq k_A \leq 2^{\ell-1} + 1 < N$ never exceeds N in the main loop. k_A cannot be zero or N; therefore, $P + Q = \mathcal{O}$ does not occur in the main loop.

As for the exceptional addition $P + P$, we must prove $2A \neq R[k_i]$. If $2A = R[k_i]$, we have two cases: $2A = R[0] = -2P$, which means that $A = -P = (N-1)P$ should be updated in a loop; $2A = R[1] = -P$, which means that $A = \frac{(N-1)}{2}P$ should be updated in a loop. However, both contradict the fact that $2 \leq k_A \leq 2^{\ell-1} + 1 < N$.

In the final correction, noting that $A \leftarrow 2A + R[0]$ can be computed without the exceptional addition of $\mathcal{O} + P$ is easy. The exceptional addition of $P + Q = \mathcal{O}$ does not appear here. If $A \leftarrow 2A + R[0] = \mathcal{O}$, then $\mathcal{O} - 2P = (N-2)P$ when $k_0 = 0$ or $\mathcal{O} - P = (N-1)P$ when $k_0 = 1$ is computed in the next step. However, if the result is $(N-2)P$ ($k_0 = 1$ because N is odd), $(N-1)P - P$ is computed in Step 6, not $\mathcal{O} - 2P = (N-2)P$; if the result is $(N-1)P$ ($k_0 = 0$ because N is odd), $(N+1)P - 2P$ is computed in Step 6, not $\mathcal{O} - P = (N-1)P$. We have a contradiction. The exceptional addition of $A \leftarrow 2A + R[0] = \mathcal{O}$ does not appear here.

The exceptional addition of $P + P$ does not appear in Step 5. If $A \leftarrow 2A + R[0] = (N-2)P + (N-2)P = (N-4)P$ is computed in Step 5, then $(N-4)P - 2P = (N-6)P$ when $k_0 = 0$ or $(N-4)P - P = (N-5)P$ when $k_0 = 1$ is computed in the next step. However, if the result is $(N-6)P$ ($k_0 = 1$ because N is odd), $(N-5)P - P$ is computed in Step 6; if the result is $(N-5)P$ ($k_0 = 0$ because N is odd), $(N-3)P - 2P$ is computed in Step 6. We have a contradiction.

$A \leftarrow A + R[k_0]$ is computed without the exceptional addition of $\mathcal{O} + P$. $R[k_0] \neq \mathcal{O}$ is the fixed elliptic curve point, and A is not updated as $\mathcal{O}$ in the previous step, as we have already shown. We can avoid the exceptional addition $A + R[k_0] = \mathcal{O}$ when $k = 0$ by applying the extended affine addition formulae. Specifically, we investigate the exceptional addition of $P+P$, $(N-2)P+(N-2)P$ when $k_0 = 0$ and $(N-1)P+(N-1)P$ when $k_0 = 1$. The computation $(N-2)P + (N-2)P$ does not appear because the k_0 of the result of $(N-2)P + (N-2)P =$

Algorithm 35 EF 4-ary LR algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $k_\ell \in \{0, 1\}$
Output: kP
Uses: A, $R[0]$, $R[1]$, $R[2]$, $R[3]$
Initialization
1: $R[1] \leftarrow -P$
2: $R[0]$, $R[2] \leftarrow DQ(R[1])$
3: $R[3] \leftarrow R[0]$
4: $A \leftarrow -R[0]$
Main loop
5: **for** $i = \ell - 1$ to 1 **do**
6: $A \leftarrow Quad(A)$
7: $A \leftarrow A + R[k_i + 2] + R[k_{i-1}]$
8: $i = i - 2$
9: **end for**
Final correction
10: $A \leftarrow 2A + R[0]$
11: $A \leftarrow A + R[k_0]$
12: $A \leftarrow (-1)^{k_\ell} \times A$
13: **return** A

$(N-4)P$ is '1' because of the odd N. Unfortunately, the computation $(N-1)P + (N-1)P$ can appear in Step 6. Our algorithms can avoid this exceptional addition when $\frac{N}{2} < N - 2$, meaning that $N > 4$.

The proof of Alg. 35 is the same as that described above. □

Alg. 34 combined with Alg. 25 (or Alg. 26) can reduce the two inversions to one for every bit computation in the main loop. On the other hand, Alg. 35 combined with the quad-algorithm and Alg. 28 computes one inversion for every bit computation with half iterations.

To improve the efficiency by reducing the number of inversions in the main loop of Alg. 35 to one, we propose an extended exception-free 4-ary LR algorithm (Alg. 36) called the EX-EF 4-ary LR algorithm. This algorithm uses the quadruple-add algorithm (Alg. 29) in the main loop, which reduces the number of inversions to one for every bit computation. In Alg. 36, the initialization of $R[0] = -6P$, $R[1] = -5P$, $R[2] = -4P$, $R[3] = -3P$, and $A = 2P$ is done without exceptional computations if $N > 6$. The idea is to replace $R[k_i + 2] + R[k_{i-1}]$ in Step 7 of Alg. 35 with $R[2k_i + k_{i-1}]$ in Step 4 of Alg. 36 to make the Montgomery trick useful without introducing excessive multiplications. Algs. 35 and 36 output the same A value at the end of the main loop. Thus, Alg. 36 can use the same final correction as Alg. 35 by including Step 6, which is the same as Steps 1–2 in Alg. 35. Alg. 36 can be regarded as an improved version of Joye's regular 4-ary LR algorithm. Alg. 36 fulfills the *Secure generality* and avoids all exceptional computations of the affine formulae when $k \in [-\frac{N}{2}, \frac{N}{2}]$, as in Alg. 35. The proof of Theorem 5.1 can easily be extended to Alg. 36.

Algorithm 36 EX-EF 4-ary LR algorithm

Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, $k_\ell \in \{0, 1\}$

Output: kP

Uses: A, $R[0]$, $R[1]$, $R[2]$, $R[3]$

Initialization

1: $\{A, R[2]\} \leftarrow DQ(P)$, $R[1] \leftarrow -(R[2] + P)$, $R[0] \leftarrow -(R[2] + A)$

2: $R[3] = P \leftarrow -(P + A)$, $R[2] \leftarrow -R[2]$

Main loop

3: **for** $i = \ell - 1$ to 1 **do**

4: $\quad A \leftarrow QA(A, R[2k_i + k_{i-1}])$ (Alg. 29), $i = i - 2$

5: **end for**

Final correction

6: $R[1] \leftarrow R[1] - R[2]$, $R[0] \leftarrow R[3] - R[1]$

7: $A \leftarrow 2A + R[0]$, $A \leftarrow A + R[k_0]$, $A \leftarrow (-1)^{k_\ell} \times A$

8: **return** A

Chapter 6

Evaluations for Our RL and LR ECSMs

In this chapter, we show the evaluation results of proposed SCAs resistant and compact ECSMs in Chapters 4 and 5. The theoretical evaluation results are shown in Section 6.1 and the experimental evaluation results are shown in Section 6.2.

6.1 Theoretical Analysis

We evaluate the performance of Algs. 23–24 and 33–36 with (extended) affine and affine combination-addition algorithms in comparison to the Montgomery ladder (Alg. 16) with CA formulae, Joye's regular 2-ary LR algorithm (Alg. 18) with CA formulae, and Joye's regular 2-ary RL algorithm with CA formulae. Table 6.1 presents the results sorted by the least amount of memory. It is worth noting that Alg. 16 leaks the last consecutive 0 bits of a scalar and dose not satisfy the secure generality, shown in Theorem 4.3.

Table 6.1: Comparison of computational costs

ECSM	Computational cost	$S = M$, $m_a = m_b = A = 0$
2-ary RL + (extended) affine + Alg. 30	$(7\ell+9)M + (3\ell+4)S + (\ell+5)I + (15\ell+22)A$	$(10\ell+13)M + (\ell+5)I$
2-ary RL + (extended) affine	$(4\ell+12)M + (3\ell+4)S + (2\ell+4)I + (12\ell+25)A$	$(7\ell+16)M + (2\ell+4)I$
Two-bit 2-ary RL + (extended) affine	$(6\ell+16)M + (5\ell+8)S + (1.5\ell+4.5)I + (15\ell+34)A$	$(11\ell+24)M + (1.5\ell+4.5)I$
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 31	$(7.5\ell+17.5)M + (5\ell+12)S + (\ell+9)I + (16\ell+49)A$	$(12.5\ell+29.5)M + (\ell+9)I$
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 32	$(10.5\ell+8.5)M + (5\ell+12)S + (0.5\ell+10.5)I + (17\ell+46)A$	$(15.5\ell+20.5)M + (0.5\ell+10.5)I$
Joye's RL [Joy09] + CA [RCB16]	$(\ell+1)(24M + 6m_a + 4m_b + 46A)$	$24(\ell+1)M$
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 25	$(9\ell+3)M + (2\ell+4)S + (\ell+3)I + (15\ell+10)A$	$(11\ell+7)M + (\ell+3)I$
EF 2-ary LR (Alg. 34) + (extended) affine	$(4\ell+8)M + (3\ell+3)S + (2\ell+2)I + (12\ell+13)A$	$(7\ell+11)M + (2\ell+2)I$
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 26	$(9\ell+3)M + (5\ell+1)S + (\ell+3)I + (17\ell+8)A$	$(14\ell+4)M + (\ell+3)I$
EF 4-ary LR (Alg. 35) + (extended) affine + Alg. 28	$(8.5\ell+9.5)M + (6\ell+6)S + (\ell+3)I + (16\ell+21)A$	$(14.5\ell+15.5)M + (\ell+3)I$
EF 4-ary LR (Alg. 35) + (extended) affine	$(6\ell+12)M + (5\ell+7)S + (1.5\ell+2.5)I + (15\ell+22)A$	$(11\ell+19)M + (1.5\ell+2.5)I$
EX-EF 4-ary LR (Alg. 36) + (extended) affine + Alg. 29	$(9\ell+19)M + (0.5\ell-0.5)m_a + (7\ell+10)S + (0.5\ell+8.5)I + (20\ell+47)A$	$(16\ell+29)M + (0.5\ell+8.5)I$
Joye's LR (Alg. 18) [Joy09] + CA [RCB16]	$\ell(24M + 6m_a + 4m_b + 46A)$	$24\ell M$
The Montgomery ladder [JY02] + CA [RCB16]	$\ell(24M + 6m_a + 4m_b + 46A)$	$24\ell M$

We evaluate memory usage by counting the number of $\mathbb{F}_p$ elements used in an ECSM, including the prime number p, elliptic curve parameters, an input scalar, an input point, intermediate memory used in scalar multiplication, and affine formulae. In terms of computational cost, we estimate the number of modulo multiplications (M), modulo squaring (S), modulo multiplications with parameters a and b (m_a and m_b, respectively), additions (A), and inversions (I). We assumed that $S = M$ and that ℓ is the bit length of $|k|$. To compare the modular inversion cost to the modular multiplication cost, we calculate the ratio r, such that $I = rM$.

Table 6.2: Comparison of memory costs

ECSM	#I/bit	Memory
2-ary RL + (extended) affine + Alg. 30	1	14
2-ary RL + (extended) affine	2	14
Two-bit 2-ary RL + (extended) affine	1.5	16
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 31	1	20
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 32	0.5	21
Joye's RL [Joy09] + CA [RCB16]	0	22
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 25	1	12
EF 2-ary LR (Alg. 34) + (extended) affine	2	12
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 26	1	13
EF 4-ary LR (Alg. 35) + (extended) affine + Alg. 28	1	15
EF 4-ary LR (Alg. 35) + (extended) affine	1.5	15
EX-EF 4-ary LR (Alg. 36) + (extended) affine + Alg. 29	0.5	19
Joye's LR (Alg. 18) [Joy09] + CA [RCB16]	0	19
The Montgomery ladder [JY02] + CA [RCB16]	0	19

Ignore the computational costs of m_a, m_b, and A and assume $S = M$. Note that ignoring the computational costs of m_a, m_b, and A is a disadvantage because ECSMs with CA formulae use significantly more m_a, m_b, and A. Table 6.1 highlights some key points about the comparison of the ECSM algorithms that 1) with the same number of inversions for every bit of computation, LR ECSM generally has lower memory requirements than the RL ECSM. 2) the computational cost of the LR ECSM is lower than that of the RL ECSM when the Montgomery trick is not used. 3) when the Montgomery trick is applied, the LR ECSM requires more multiplications and squares than the RL ECSM, as the computations in the main loop of the LR ECSM are dependent on each other. 4) The larger the ratio r, the more efficient the ECSM is with fewer inversions.

Table 6.3 lists the most efficient ECSM algorithm based on the ratio $r = \frac{I}{M}$. Even when the sufficiently large bit-length of the scalar, ℓ, changes, the conditions for efficiency remain the same. In situations where $m_a = A = 0$, such as in the NIST elliptic curves in Table 6.4, the interval of r where our algorithms are more efficient becomes larger. Specifically, if $r < \frac{3\ell+2}{\ell-3}$ ($r < 3.04$ when $\ell = 256$), then the EF 2-ary LR (Alg. 34) with (extended) affine is the most efficient; if $\frac{3\ell+2}{\ell-3} < r < \frac{11\ell-15}{\ell-11}$ ($3.04 < r < 11.43$ when $\ell = 256$), then the 2-ary RL with Alg. 30 and (extended) affine is the most efficient; if $\frac{11\ell-15}{\ell-11} < r < \frac{25\ell-41}{\ell+21}$ ($11.43 < r < 22.96$ when $\ell = 256$), then the EX-EF 4-ary RL (Alg. 33) with Alg. 32 and (extended) affine is the most efficient. Joye's LR (Alg. 18) with CA formulae is more efficient only when $r > \frac{25\ell-41}{\ell+21}$ ($r > 22.96$ when $\ell = 256$).

For memory usage, (Alg. 34) and (Alg. 34 with Alg. 25) use the least amount of memory for the 12 field elements, reducing that of the Montgomery ladder with CA formulae and Joye's LR with CA formulae by 36.84%, that of 2-ary RL with (extended) affine by 14.29% and that of Joye's RL with CA formulae by 45.45%.

Table 6.3: Most efficient ECSMs with the condition of $r = \frac{I}{M}$ ($m_a = m_b = A = 0$, $S = M$, and ℓ is the bit length of $|k|$.)

ECSM	**Condition**	**Memory**
EF 2-ary LR (Alg. 34) + (extended) affine	$r < \frac{3\ell+2}{\ell-3}$	12
2-ary RL + (extended) affine + Alg. 30	$\frac{3\ell+2}{\ell-3} < r < \frac{11\ell-15}{\ell-11}$	14
EX-EF 4-ary RL (Alg. 33) + (extended) affine + Alg. 32	$\frac{11\ell-15}{\ell-11} < r < \frac{17\ell-41}{\ell+21}$	21
Joye's LR (Alg. 18) + CA	$r > \frac{17\ell-41}{\ell+21}$	19

6.2 Experimental Analysis

We implemented all ECSMs listed in Table 6.1 on the NIST-224, NIST-256, and NIST-384 elliptic curves. Table 6.4 lists these elliptic curves. We randomly generated 10^4 test scalars within the interval $[-\frac{N}{2}, \frac{N}{2}]$, where N is the order of the point P, to measure the average scalar multiplication time of these algorithms. We used a C programming language with the GNU MP 6.1.2 library for multiple precision arithmetic on an Intel (R) Core (TM) i7-8650U CPU @ 1.90 GHz 2.11 GHz personal 64-bit computer with 16.0 GB of RAM and the Ubuntu 20.04.1 LTS operating system. To ensure fair comparison, we turned off the Intel turbo boost feature during the experiments and made our computer working on the OCTA core processor at 1.80 GHz.

Table 6.4: NIST elliptic curves $y^2 = x^3 - 3x + b$

NIST-224	$b = 18958286285566608000408668544493926415504680968679321075787234672564$
NIST-256	$b = 41058363725152142129326129780047268409114441015993725554835256314039467401291$
NIST-384	$b = 27580193559959705877849011840389048093056905856361568521428707301988689241309860865136260764883745107765439761230575$

Table 6.5 shows the average time results of our ECSMs implementation. It can be observed that the EX-EF 4-ary LR algorithm (Alg. 36) with (extended) affine combined with Alg. 29 is the most efficient for NIST-224, NIST-256, and NIST-384. It outperforms the other ECSMs in terms of average time,

- Reducing the computation time of Joye's LR with CA [Joy09, RCB16] by 34.41% for NIST-224, 29.9% for NIST-256, and 25.17% for NIST-384, respectively.
- Reducing the computation time of the Montgomery ladder with CA [JY02] by 34.48% for NIST-224, 30.42% for NIST-256 and 25.72% for NIST-384, respectively.
- Reducing the computation time of the two-bit 2-ary RL with (extended) affine by 25.1% for NIST-224, 27.14% for NIST-256, and 31.68% for NIST-384, respectively.

The computation time of the EX-EF 4-ary LR (Alg. 36) with (extended) affine combined with Alg. 29 and the EX-EF 4-ary RL (Alg. 33) with (extended) affine combined with Alg. 32 is

almost the same. The EX-EF 4-ary LR (Alg. 36) with (extended) affine combined with Alg. 29 uses 2 less memories.

Table 6.5: Average computation time for one scalar multiplication (microseconds (μs))

ECSM	NIST-224	NIST-256	NIST-384	Memory	Secure generality
2-ary RL + (extended) affine + Alg. 30	1529.59	1825.53	3784.15	14	True
2-ary RL + (extended) affine	2108.69	2667.63	5656.22	14	True
Two-bit 2-ary RL + (extended) affine	1955.39	2394.81	5029.17	16	True
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 31	1706.95	1994.24	4170.16	20	True
EX-EF 4-ary RL (Alg. 33) + (extended) affine +Alg. 32	1510.89	1748.62	3437.06	21	True
Joye's RL [Joy09] + CA [RCB16]	2247.93	2513.67	4638.60	22	True
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 25	1543.48	1852.41	3853.34	12	True
EF 2-ary LR (Alg. 34) + (extended) affine	2108.18	2647.64	5649.23	12	True
EF 2-ary LR (Alg. 34) + (extended) affine + Alg. 26	1711.99	2034.72	4249.43	13	True
EF 4-ary LR (Alg. 35) + (extended) affine + Alg. 28	1782.17	2096.95	4350.95	15	True
EF 4-ary LR (Alg. 35) + (extended) affine	1953.89	2392.44	5014.68	15	True
EX-EF 4-ary LR (Alg. 36) + (extended) affine + Alg. 29	1464.66	1744.81	3436.05	19	True
Joye's LR (Alg. 18) [Joy09] + CA [RCB16]	2233.13	2488.99	4591.66	19	True
The Montgomery ladder [JY02] + CA [RCB16]	2235.31	2507.66	4625.93	19	False

The efficiency of our algorithm is determined by the ratio $r = \frac{I}{M}$, as previously mentioned. We conducted experiments by randomly generating 6.4×10^6 numbers for 224, 256, and 384 bits and measuring the average computation time for M, I, and I/M, respectively, as shown in Table 6.6. In the case of the NIST elliptic curves, without ignoring m_b, the EX-EF 4-ary LR (Alg. 36) with (extended) affine combined with Alg. 29 is the most efficient when $12.53 < r < 22.07$ for 224 bits, $12.47 < r < 22.29$ for 256 bits, and $12.31 < r < 22.84$ for 384 bits. Therefore, the EX-EF 4-ary LR (Alg. 36) with (extended) affine combined with Alg. 29 is the most efficient in our experiments. These findings are consistent with the theoretical analysis presented in Table 6.1.

To conclude, LR ECSMs are an efficient solution for applications where memory constraints are a concern, such as IoT devices. These algorithms can be executed with minimal memory requirements.

Table 6.6: Ratio of computational costs of inversion to modulo multiplication (μs)

	I	M	I/M
224 bits	0.23	3.30	14
256 bits	0.20	3.80	19
384 bits	0.29	5.67	20

Chapter 7

Compact and Constant-Time GCD and Modular Inversion

In this chapter, we consider compact and constant-time GCD and modular inversion with good characteristics of short iterations and simple computations during one iteration. In Section 7.1, we combine the basic concept of BEA and Lemma 7.1 and define our iteration formulae. The convergence and required number of iterations of the iteration formulae are proved. New short-iteration GCD (SI-GCD) and modular inversion (SI-MI) algorithms, whose computations in each branch are balanced are proposed based on the iteration formulae and the proofs. In Section 7.2, we propose CT-GCD and CTMI algorithms, called short-iteration CT-GCD (SICT-GCD) and short-iteration CTMI (SICT-MI), that have short iterations and simple computations. The theoretical evaluation results are shown in Section 7.3 and the experimental evaluation results are shown in Section 7.4.

7.1 Iteration Formulae and Proofs

Let us start from a simple lemma of GCD. Lemma 7.1 implies that the GCD of any two of a, b, and $a-b$, where $a, b \in \mathbb{Z}$ and $a \geq b \geq 0$, is the same. Lemma 7.1 is used to demonstrate Lemma 7.2.

Lemma 7.1. $\gcd(a,b) = \gcd(b, a-b) = \gcd(a, a-b)$, *where* $a, b \in \mathbb{Z}$ *and* $a \geq b \geq 0$.

Proof. With the well-known fact that $\gcd(a,b) = \gcd(a-nb, b)$ for any n, it is clear that $\gcd(a,b) = \gcd(a-b, b)$ with $n = 1$ and $\gcd(b, a-b) = \gcd(a, a-b)$ with $n = -1$. □

Based on the basic concept of BEA in Equation (2.3) and Lemma 7.1, we show that GCD has the following equivalence relations:

Lemma 7.2. *GCD has the following equality.*

$$\gcd(a,b) = \begin{cases} \gcd((a-b)/2, b), & a \text{ and } b \text{ are odd} \\ \gcd(a/2, a-b), & a \text{ is even and } b \text{ is odd} \\ \gcd(b/2, a-b), & a \text{ is odd and } b \text{ is even.} \end{cases}$$

$a, b \in \mathbb{Z}$, $a \geq b \geq 0$ and a and/or b are/is odd.

Proof.
- Assume that a and b are odd. $\gcd(a, b) = \gcd(b, a-b)$ by Lemma 7.1 and $\gcd(b, a-b) = \gcd(b, (a-b)/2)$ by the concept of BEA.
- Assume that a is even and b is odd. $\gcd(a, b) = \gcd(a, a-b)$ by Lemma 7.1 and $\gcd(a, a-b) = \gcd(a/2, a-b)$ by the concept of BEA.
- Assume that a is odd and b is even. $\gcd(a, b) = \gcd(b, a-b)$ by Lemma 7.1 and $\gcd(b, a-b) = \gcd(b/2, a-b)$ by the concept of BEA.

□

Lemma 7.2 gave rise to the iteration formula, which is the core computation in the proposed SICT-GCD and SICT-MI.

Definition 7.1. *The iteration formula $(a_{n+1}, b_{n+1}) = f(a_n, b_n)$ for $a_n, b_n \in \mathbb{Z}$, $a_n \geq b_n \geq 0$ and a_n and/or b_n are/is odd, is defined as follows:*

$$f(a_n, b_n) = \begin{cases} \texttt{Branch}_1 : \texttt{max.min}((\texttt{a}_\texttt{n} - \texttt{b}_\texttt{n})/2, \texttt{b}_\texttt{n}), a_n \text{ and } b_n \text{ are odd} \\ \texttt{Branch}_2 : \texttt{max.min}(\texttt{a}_\texttt{n}/2, \texttt{a}_\texttt{n} - \texttt{b}_\texttt{n}), a_n \text{ is even, } b_n \text{ is odd} \\ \texttt{Branch}_3 : \texttt{max.min}(\texttt{b}_\texttt{n}/2, \texttt{a}_\texttt{n} - \texttt{b}_\texttt{n}), a_n \text{ is odd, } b_n \text{ is even.} \end{cases}$$

Definition 7.2. $(a, b) = \texttt{max.min}(a, b)$, *where $a, b \in \mathbb{Z}$, is defined as follows:*

$$\texttt{max.min}(a, b) = \begin{cases} (a, b) \text{ when } a \geq b \\ (b, a) \text{ otherwise.} \end{cases}$$

Lemma 7.3. *Let $a_n = d \in \mathbb{Z}^+$ and $b_n = 0$, where d is odd, be the inputs of f in Definition 7.1. Then, regardless of the number of f iterations, the outputs are the same as the inputs, which are $a_{n+i} = d$, $b_{n+i} = 0$, $i \geq 1$. Therefore, $f(f \cdots f(f(a_n, b_n))) = (a_n, b_n)$.*

Proof. Assume that $a_n = d \in \mathbb{Z}^+$ and $b_n = 0$, where d is odd. $a_{n+1} = d$ and $b_{n+1} = 0$ are computed using $\texttt{Branch}_3$, which are the same as the inputs. □

Lemma 7.4. *The sequence $\{\mathbb{Z} \ni (a_i + b_i) > 0\}$ is a monotonically decreasing sequence for every two steps, where a_0 and b_0 are the initial inputs, and a_i and b_i ($\mathbb{Z} \ni i > 0$) are updated by the iteration formula in Definition 7.1.*

Proof. Every two steps can be classified into nine patterns of $(\texttt{Branch}_\texttt{i}, \texttt{Branch}_\texttt{j})$, where $i, j \in \{1, 2, 3\}$, as listed in Table 7.1. All patterns are proven by (1)–(7).

(1) After $\texttt{Branch}_1$, $a_{i+1} + b_{i+1} < a_i + b_i$ holds in the Equation (7.1).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{a_i - b_i}{2} + b_i) - (a_i + b_i) \\ &= -\frac{a_i + b_i}{2} < 0 \end{aligned} \tag{7.1}$$

(2) After $\texttt{Branch}_2$ when $4b_i > a_i \geq b_i$, $a_{i+1} + b_{i+1} < a_i + b_i$ holds in Equation (7.2).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{a_i}{2} + a_i - b_i) - (a_i + b_i) \\ &= \frac{a_i - 4b_i}{2} < 0 \end{aligned} \tag{7.2}$$

(3) After $\texttt{Branch}_2$ when $a_i \geq 4b_i$, $a_{i+2} + b_{i+2} < a_i + b_i$. $a_{i+1} = a_i - b_i, b_{i+1} = a_i/2$ are updated because of $a_i - b_i - a_i/2 = (a_i - 2b_i)/2 > 0$. a_{i+1} is odd because a_i is even and b_i is odd. Then a_{i+2}, b_{i+2} are computed by $\texttt{Branch}_1$ and $a_{i+2} + b_{i+2} = (3a_i - 2b_i)/4$, when $a_i/2$ is odd. a_{i+2}, b_{i+2} are computed by $\texttt{Branch}_3$ and $a_{i+2} + b_{i+2} = (3a_i - 4b_i)/4$, when $a_i/2$ is even. Finally, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Equation (7.3).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= -\frac{a_i + 6b_i}{4} < 0 \text{ when } a_i/2 \text{ is odd.} \\ &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= -\frac{a_i + 8b_i}{4} < 0 \text{ when } a_i/2 \text{ is even} \end{aligned} \tag{7.3}$$

(4) After $\texttt{Branch}_3$, $a_{i+1} + b_{i+1} \leq a_i + b_i$ holds in Equation (7.4).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{b_i}{2} + a_i - b_i) - (a_i + b_i) \\ &= -\frac{3b_i}{2} \leq 0 \ (=0 \text{ when } b_i = 0) \end{aligned} \tag{7.4}$$

(5) After $(\texttt{Branch}_1, \texttt{Branch}_2)$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Equation (7.5).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= (\frac{a_i - b_i}{4} + \frac{a_i - 3b_i}{2}) - (a_i + b_i) \\ &= -\frac{a_i + 11b_i}{4} < 0 \end{aligned} \tag{7.5}$$

(6) After $(\texttt{Branch}_3, \texttt{Branch}_2)$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Equation (7.6).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= (\frac{b_i}{4} + \frac{3b_i - 2a_i}{2}) - (a_i + b_i) \\ &= \frac{3b_i - 8a_i}{4} < 0 \end{aligned} \tag{7.6}$$

(7) After $(\texttt{Branch}_2, \texttt{Branch}_2)$ when $2b_i \geq a_i \geq b_i$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Equation (7.7).

Note that there is no pattern of $(\texttt{Branch}_2, \texttt{Branch}_2)$ when $a_i > 2b_i$.

$$\begin{aligned}
&(a_{i+2} + b_{i+2}) - (a_i + b_i) \\
&= (\frac{a_i}{4} + \frac{2b_i - a_i}{2}) - (a_i + b_i) \\
&= -\frac{5a_i}{4} < 0
\end{aligned} \tag{7.7}$$

In summary, all cases are proven as shown in Table 7.1. □

Table 7.1: Proved case

Case	Proved by
$(\texttt{Branch}_1, \texttt{Branch}_1)$	(1)
$(\texttt{Branch}_1, \texttt{Branch}_2)$	(5)
$(\texttt{Branch}_1, \texttt{Branch}_3)$	(1) and (4)
$(\texttt{Branch}_2, \texttt{Branch}_1)$	(1), (2) and (3)
$(\texttt{Branch}_2, \texttt{Branch}_2)$	(7)
$(\texttt{Branch}_2, \texttt{Branch}_3)$	(2), (3) and (4)
$(\texttt{Branch}_3, \texttt{Branch}_1)$	(1) and (4)
$(\texttt{Branch}_3, \texttt{Branch}_2)$	(6)
$(\texttt{Branch}_3, \texttt{Branch}_3)$	(4)

Theorem 7.1. *(Convergence) After sufficient iterations of the iteration formula defined in Definition 7.1, any valid inputs a_0 and b_0 converge to $a_n = d$ and $b_n = 0$, where $d = \gcd(a_0, b_0)$ and d is odd because a_0 and/or b_0 are/is odd.*

Proof. By Definition 7.1 and Lemmas 7.2, 7.3, and 7.4, Theorem 7.1 can be proven. □

As shown in Theorem 7.1, after sufficient iterations of the iteration formula, $a_n = \gcd(a_0, b_0)$ and $b_n = 0$ are the outputs. Then, we construct our SICT-GCD and SICT-MI by determining the necessary number of iterations.

We have already shown our iteration formula in Definition 7.1, which has simple and identical computations (a shift and a subtraction) in each iteration. According to Theorem 7.2, the iteration formula converges after $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ iterations. Thus, the number of iterations are 448, 512, and 768 for 224-, 256-, and 384-bit GCD computations and modular inversions, respectively. Our number of iterations is fewer than that of hdBY [PGrb21] and the same as that of BOS [Bos14]. The required number of iterations is not intuitive in our iteration formula because the decrease in the total bit length of the inputs a_i and b_i after $\texttt{Branch}_2$ cannot be observed directly.

Lemma 7.5. *For any valid a_i and b_i, $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least one after $\texttt{Branch}_1$, $\texttt{Branch}_2$ $(2b_i \geq a_i \geq b_i)$ and $\texttt{Branch}_3$ of the iteration formula in Definition 7.1.*

Proof. (1) It is clear that $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least one after $\texttt{Branch}_1$ or $\texttt{Branch}_3$ of the iteration formula in Definition 7.1.

(2) Assume that even a_i has x bits, odd b_i has y bits, and $2b_i \geq a_i \geq b_i$. After $\texttt{Branch}_2$ of the iteration formula, $a_{i+1} = a_i/2$ has $x-1$ bits and $0 \leq b_{i+1} = a_i - b_i \leq b_i$ has y bits at most. Thus, $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least one after $\texttt{Branch}_2$ ($2b_i \geq a_i \geq b_i$). □

Lemma 7.6. *Assume that even a_i has x bits, odd b_i has y bits, and $a_i > 2b_i$, $x - y \geq 1$. Then $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least $x - y + 1$ bits after $x - y + 1$ iterations.*

Proof. a_{i+1} and b_{i+1} are computed by $\texttt{Branch}_2$ firstly. $a_{i+1} = a_i - b_i$ and $b_{i+1} = a_i/2$ because $a_i - b_i - a_i/2 = (a_i - 2b_i)/2 > 0$. When $x - y = 1$, an additional iteration is considered. If $a_i/2$ is odd, then $a_{i+2} = a_i/2$ with $x-1$ bits and $b_{i+2} = (a_i - 2b_i)/4$ with $x-2$ bits at most by $\texttt{Branch}_1$. If $a_i/2$ is even, then $a_{i+2} = a_i/4$ with $x-2$ bits and $b_{i+2} = (a_i - 2b_i)/2$ with at most $x-1$ bits by $\texttt{Branch}_3$. Subsequently, $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by $x - y + 1 = 2$ bits at least.

When $x-y = 2$, take a_{i+3} and b_{i+3} into consideration. From the computations, $\texttt{bitlen}(a_{i+3}) + \texttt{bitlen}(b_{i+3})$ is at most $(x-2) + (x-3) = x + y - 3$. Thus, $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least $x - y + 1 = 3$ bits after $x - y + 1 = 3$ iterations.

For $x-y = 3 \cdots$, one can continue to calculate a_{i+4} and $b_{i+4} \cdots$, and find that $\texttt{bitlen}(a_{i+k}) + \texttt{bitlen}(b_{i+k})$ is at most $(x-k) + (x-k+1) = x + y - k$, where $x - y = k - 1$. Subsequently, $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least $x - y + 1$ bits after $x - y + 1$ iterations. □

Theorem 7.2. *After $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ iterations of the iteration formula defined in Definition 7.1, the valid inputs (a_0, b_0) converge to $(a_n = d, b_n = 0)$, where $\gcd(a_0, b_0) = d$ and d is odd.*

Proof. Theorem 7.1 shows that the valid inputs (a_0, b_0) converge to $(a_n = d, b_n = 0)$, where $\gcd(a_0, b_0) = d$ and d is odd with sufficient iterations. By Lemmas 7.5 and 7.6, $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ is reduced by at least one after each iteration on average. Thus, the iteration formula converges after at most $bitlen(a_0) + bitlen(b_0)$ iterations. □

Using the iteration formula in Definition 7.1 and the required number of iterations, the SI-GCD algorithm is shown in Alg. 37. It is not difficult to determine the transition matrices for a_i and b_i. The SI-MI algorithm is presented according to the transition matrices in Alg. 38. The computations for each iteration consist of two shifts and two subtractions in Algorithm 38. Note that all the transition matrices are multiplied by two to eliminate multiplications by $1/2$. Thus, we need to multiply the result by $2^{-l} \mod p$, which can be precomputed, where l is the number of iterations.

Montgomery inversion, $a^{-1} \times R \bmod p$, can be computed using SI-MI, where R is generally $2^{\texttt{bitlen}(p)}$ [Mon85]. By setting $pre\_com = 2^{-\texttt{bitlen}(p)} \bmod p$, the output of Algorithm 38 is $a^{-1} \times 2^{\texttt{bitlen}(p)} \bmod p$. For different choices of R, one can change pre_com and obtain $a^{-1} \times R \bmod p$ using Algorithm 38.

For clarity, the computational flow in one iteration of Algorithm 38 can be described as follows:

Algorithm 37 SI-GCD

Input: $a, b \in \mathbb{Z}$, $a \geq b \geq 0$, a or b is odd. $l = 2 \cdot \texttt{bitlen}(a)$.
Output: $v = \gcd(a, b)$
Initialization:
1: $u = b$; $v = a$;
2: **for** $i = 0$ to $l - 1$ **do**
3: $t_1 = v - u$;
4: **if** u is even **then**
5: $u = u >> 1$; $(v, u) = \texttt{max.min}(u, t_1)$;
6: **else**
7: **if** v is odd **then**
8: $t_1 = t_1 >> 1$; $(v, u) = \texttt{max.min}(t_1, u)$;
9: **else**
10: $v = v >> 1$; $(v, u) = \texttt{max.min}(v, t_1)$;
11: **end if**
12: **end if**
13: **end for**
14: **return** v

(1) Update $t_1 = v - u$ and $t_2 = q - r$.

(2) According to the LSB of u and the LSB of v, select the $\texttt{Branch}_\texttt{i}$.

(3) According to the selected $\texttt{Branch}_\texttt{i}$, right shift the data in the second column of Table 7.2 and left shift the data in the third column of Table 7.3.

(4) According to the selected $\texttt{Branch}_\texttt{i}$, compare the data in the second column of Table 7.2 and the data in the third column of Table 7.2. Then update u, v, q, and r by the results of the comparison.

Table 7.2: Data matrix of u and v.

$\texttt{Branch}_1$	t_1	u
$\texttt{Branch}_2$	v	t_1
$\texttt{Branch}_3$	u	t_1

Table 7.3: Data matrix of q and r.

$\texttt{Branch}_1$	t_2	r
$\texttt{Branch}_2$	q	t_2
$\texttt{Branch}_3$	r	t_2

Algorithm 38 SI-MI

Input: $a, p \in \mathbb{Z}$, $p > a > 0$, $\gcd(a, p) = 1$. $l = 2 \cdot \texttt{bitlen}(p)$. $pre\_com = 2^{-l} \bmod p$.
Output: $q = a^{-1} \bmod p$
Initialization:
1: $u = a$; $v = p$; $q = 0$; $r = 1$;
2: **for** $i = 0$ to $l - 1$ **do**
3: $t_1 = v - u$; $t_2 = q - r$;
4: **if** u is odd, and v is odd **then**
5: $t_1 = t_1 >> 1$; $r = r << 1$;
6: **if** $u > t_1$ **then**
7: $q = r$; $r = t_2$; $v = u$; $u = t_1$;
8: **else**
9: $q = t_2$; $r = r$; $v = t_1$; $u = u$;
10: **end if**
11: **else if** u is odd, and v is even **then**
12: $v = v >> 1$; $t_2 = t_2 << 1$;
13: **if** $t_1 > v$ **then**
14: $r = q$; $q = t_2$; $u = v$, $v = t_1$;
15: **else**
16: $q = q$; $r = t_2$; $v = v$; $u = t_1$;
17: **end if**
18: **else**
19: $u = u >> 1$; $t_2 = t_2 << 1$;
20: **if** $t_1 > u$ **then**
21: $r = r$; $q = t_2$; $u = u$; $v = t_1$;
22: **else**
23: $q = r$; $r = t_2$; $v = u$; $u = t_1$;
24: **end if**
25: **end if**
26: **end for**
27: $q = q \times pre\_com \bmod p$
28: **return** q

7.2 Short-Iteration Constant-Time GCD and Modular Inversion

Because each branch in Algorithms 37 and 38 has the same computations, the conditional statements can be removed, as shown in Algorithm 39. Note that this is not the only way to remove conditional statements. For instance, t_1 and t_2 can also be updated as $t_1 = szu \oplus (\bar{s} + \bar{z})(v-u)$ and $t_2 = [sz(v-u) \oplus s\bar{z}v \oplus \bar{s}zu] >> 1$, using three additional XOR operations on F_p and one less addition on F_p. SICT-GCD can be easily obtained by removing the computations of q and r from Algorithm 39. Thus, we have omitted its description. Function $\texttt{cmp}(a, b)$ returns one if $a \geq b$ and returns zero if $a < b$. There are two secrets-independent table look-ups of $sort_1[2]$ and $sort_2[2]$ according to the comparison results of t_1 and t_2. Without a loss of generality, we assume that the result of the comparison is $t_2 < t_1$, which can occur in any branch. The

operational flow cannot be determined based on this information.

Algorithm 39 SICT-MI

Input: $a, p \in \mathbb{Z}$, $p > a > 0$, $\gcd(a, p) = 1$. $l = 2 \cdot \texttt{bitlen}(p)$. $pre\_com = 2^{-l} \bmod p$.
Output: $q = a^{-1} \bmod p$
Initialization:
1: $u = a$; $v = p$; $q = 0$; $r = 1$;
2: $sort_1[2] = \{t_1, t_2\}$; $sort_2[2] = \{t_3, t_4\}$;
3: **for** $i = 0$ to $l - 1$ **do**
4: $s = u_{lsb}$; $z = v_{lsb}$;
5: $t_1, t_2 = (s \oplus z)v + ((sz << 1) - 1)u, [sv + (2 - (s << 1) - z)u] >> 1$;
6: $t_3, t_4 = [(s \oplus z)q + ((sz << 1) - 1)r] << 1, sq + (2 - (s << 1) - z)r$;
7: $s = \texttt{cmp}(t_2, t_1)$; $z = !s$;
8: $v = sort_1[s]$; $u = sort_1[z]$;
9: $q = sort_2[s]$; $r = sort_2[z]$;
10: **end for**
11: $q = q \times pre\_com \bmod p$
12: **return** q

7.3 Theoretical Analysis

Table 7.4 shows the comparison between FLT-CTMI, BOS [Bos14], BY [BY19], hdBY [PGrb21] and our algorithms, where S, M, xor, add, sub, and shift represent a square, a multiplication, an xor, an addition, a subtraction, and a shift on $\mathbb{F}_p$, respectively. Table 7.5 shows the number of iterations for 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit computations of FLT-CTMI, BOS, BY, hdBY and our algorithms. FLT-CTMI can only be used to compute modular inversion over prime numbers and is inefficient for general inputs. By contrast, BOS, BY, hdBY and SICT-GCD can compute $\gcd(a, b)$, where a and/or b are/is odd. BOS, BY, hdBY and SICT-MI can compute the Montgomery inversions and modular inversion over any number if it exists. BOS has the same number of iterations as our algorithms. However, its computations, not only the computations on F_p but also the computations of control signals in one iteration, are more complicated than ours (Algorithm 2 in [Bos14]). The computations of BY and hdBY in one iteration are similar to ours. However, their number of iterations, even that of hdBY, is larger than ours. Actually, SICT-GCD and SICT-MI save 13.35%, 13.22%, 13.22%, 13.24%, 13.19%, 13.19 and 13.18% of the number of iterations for 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit computations of hdBY, respectively.

We analyze the maximum parallel data flow in one iteration of BOS, BY, hdBY, SICT-MI, as shown in Tables 7.6-7.8, where the computations in each row can be computed in parallel. Along with the assumption that an addition, a subtraction, and a shift on $\mathbb{F}_p$ are more costly and omitting the other computations, all BOS, BY, hdBY, SICT-MI use an addition and a shift on $\mathbb{F}_p$ in an iteration considering the maximum parallel data flow. However, BOS is not compact, where two subtractions, one addition, and four shifts on $\mathbb{F}_p$ are computed in parallel; BY and hdBY uses more iterations. By contrast, our algorithms are compact and with short-iteration.

Table 7.4: Comparison of CTMI.

	#iterations	Computations in one iteration
FLT-CTMI	$\texttt{bitlen}(p-2)-1$	S (or $S+M$)
BOS [Bos14]	$2\cdot\texttt{bitlen}(p)$	add + 2sub + 6shift
BY [BY19]	$\lfloor(49\texttt{bitlen}(p)+57)/17\rfloor$	4xor + 2add + 2shift
hdBY [PGrb21]	$\texttt{max}(2\lfloor(2455\log_2(M)+1402)/1736\rfloor,$ $2\lfloor(2455\log_2(M)+1676)/1736\rfloor-1),$ $M\geq 157,\ 0\leq g\leq f\leq M$	4xor + 2add + 2shift
SICT-MI	$2\cdot\texttt{bitlen}(p)$	4add + 2shift

Table 7.5: The number of iterations of CTMI for 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit computations.

	224-bit	256-bit	384-bit	511-bit	1020-bit	1790-bit	2048-bit
FLT-CTMI	223	255	383	510	1019	1789	2047
BOS [Bos14]	448	512	768	1022	2040	3580	4096
BY [BY19]	634	724	1086	1446	2886	5064	5794
hdBY [PGrb21]	517	590	885	1178	2350	4124	4718
SICT-MI	448	512	768	1022	2040	3580	4096

Table 7.6: The maximum parallel data flow in one iteration of BOS (Algorithm 2 in [Bos14]).

$uv_< = u < v;$	$uv_= = u == v;$	$u' = u - v;$
$v' = v - u;$	$\tilde{s} = s << 1;$	$\tilde{r} = r << 1;$
$rs = r + s;$	$\tilde{u} = u >> 1;$	$\tilde{v} = v >> 1;$
$u_0 = u_{lsb};$	$v_0 = v_{lsb};$	
$d = 0 - uv_=;$	$u_0 = 0 - u_0;$	$v_0 = 0 - v_0;$
$uv_< = 0 - uv_<;$	$u' = u' >> 1;$	$v' = v' >> 1;$
$m_1 = d \vee u_0;$		
$m_2 = \sim m_1;$		
$\text{select}(u, \tilde{u}, m_2, u, m_1);$	$\text{select}(s, \tilde{s}, m_2, s, m_1);$	$S = d \vee m_2;$
$m3 = S \vee v_0;$		
$m_4 = \sim m_3;$		
$\text{select}(v, \tilde{v}, m_4, v, m_3);$	$\text{select}(r, \tilde{r}, m_4, r, m_3);$	$S = S \vee m_4;$
$m_5 = S \vee uv_<;$		
$m_6 = \sim m_5;$		
$\text{select}(u, u', m_6, u, m_5);$	$\text{select}(r, rs, m_6, r, m_5);$	
$\text{select}(s, \tilde{s}, m_6, s, m_5);$	$S = S \vee m_6;$	
$m_7 = \sim S;$		
$\text{select}(v, v', m_7, v, S);$	$\text{select}(s, rs, m_7, s, S);$	
$\text{select}(r, \tilde{r}, m_7, r, S);$		

Table 7.7: The maximum parallel data flow in one iteration of BY and hdBY [BY19] (Algorithm 12).

$z = u_{lsb}$;	$s = \texttt{signbit}(\delta)$;	$t_2 = v \oplus u$;
$t_3 = q \oplus r$;		
$s = !s$;	$t_1 = zv$;	$t_4 = zq$;
$f_1 = sz$;		
$f_2 = f_1 << 1$;	$t_2 = f_1 t_2$;	$t_3 = f_1 t_3$;
$f_2 = 1 - f_2$;	$v = v \oplus t_2$;	$q = q \oplus t_3$;
$t_1 = f_2 t_1$;	$t_4 = f_2 t_4$;	$\delta = f_2 \delta$;
$\delta = 1 + \delta$;	$u = u + t_1$;	$r = t_4 + r$;
$u = u >> 1$;	$q = q << 1$;	

Table 7.8: The maximum parallel data flow in one iteration of SICT-MI (Algorithm 39).

$s = u_{lsb}$;	$z = v_{lsb}$;	
$f_1 = s \oplus z$;	$f_2 = sz$;	$f_3 = s << 1$;
$t_2 = sv$;	$t_4 = sq$;	
$f_2 = f_2 << 1$;	$f_3 = 2 - f_3$;	$t_1 = f_1 v$;
$t_3 = f_1 q$;		
$f_2 = f_2 - 1$;	$f_3 = f_3 - z$;	
$t_5 = f_2 u$;	$t_7 = f_2 r$;	$t_6 = f_3 u$;
$t_8 = f_3 r$;		
$t_1 = t_1 + t_5$;	$t_2 = t_2 + t_6$;	$t_3 = t_3 + t_7$;
$t_4 = t_4 + t_8$;		
$t_2 = t_2 >> 1$;	$t_3 = t_3 << 1$;	
$s = \texttt{cmp}(t_2, t_1)$;		
$z = !s$;		
$v = sort_1[s]$;	$u = sort_1[z]$;	$q = sort_2[s]$;
$r = sort_2[z]$;		

7.4 Experimental Analysis

Our experimental platform uses the C programming language with GNU MP 6.1.2, which is a multiple precision arithmetic library, and Intel (R) Core (TM) i7-8650U CPU @ 1.90 GHz 2.11 GHz personal 64-bit computer with 16.0 GB RAM; the operating system is Ubuntu 20.04.3 LTS. We turn off the Intel turbo boost to ensure that our computer works at 1.80 GHz.

We implement BOS, BY, hdBY, and SICT-GCD to compare the efficiency of GCD computations. In our experiments, to compute the GCD, we generate seven sets, each with 10^5 random odd numbers. The numbers in each set are 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit. We compute the GCD of the numbers in each of the sets and P224 − 1, P256 − 1, P384 − 1,

$P512-1$, $P1024-1$, $P1792-1$, $P2048-1$. Here, P224, P256, P384 are national institute of standards and technology (NIST) primes, P512, P1024, and P1792 are primes recommended for use in CSIDH [CLM+18], and P2048 is a random prime number of 2048 bits. Subsequently, the average clock cycles are measured by `rdtsc`. The results are presented in Table 7.9.

From Table 7.9, it is clear that our proposed SICT-GCD is more efficient than BOS, BY, and hdBY. SICT-GCD saves 7.44%, 13.78%, 16.67%, 18.45%, 24.77%, 27.05% and 28.3% of the clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit GCD computations, respectively.

Table 7.9: Comparison of average clock cycles for GCD computation.

	224 bits	256 bits	384 bits	511 bits	1020 bits	1790 bits	2048 bits
BOS [Bos14]	163085	190734	294382	402489	885569	1717641	2016499
BY [BY19]	107117	134523	211943	290765	667642	1328249	1591591
hdBY [PGrb21]	91997	116523	184972	256455	602059	1217655	1454694
SICT-GCD	85156	100464	154142	209138	452901	888231	1042982

We implement FLT-CTMI, BOS, BY, hdBY, and SICT-MI to compare the efficiency of modular inversion computations. Seven sets, each with 10^5 random numbers of 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit, are generated. Their modular inversions over P224, P256, P384, P512, P1024, P1792, and P2048, respectively, are computed. The average clock cycles are presented in Table 7.10.

Table 7.10: Comparison of average clock cycles for modular inversion.

	224 bits	256 bits	384 bits	511 bits	1020 bits	1790 bits	2048 bits
FLT-CTMI	206676	154477	389729	615544	2705522	10754437	14650073
BOS [Bos14]	311512	356102	559549	780402	1744711	3397150	4046376
BY [BY19]	262478	295822	481934	666601	1527635	3060803	3675057
hdBY [PGrb21]	223057	248520	400410	565481	1294404	2570412	3067074
SICT-MI	184230	208553	325659	471916	1075802	2119803	2522420

From Table 7.10, it is evident that FLT-CTMI is the most efficient for 256-bit modular inversions because of the small Hamming weight of $P256-2$, which is 128. SICT-MI is the most efficient CTMI for 224-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions. It saves 17.41%, 16.08%, 18.67%, 16.55%, 16.89%, 17.53%, and 17.76% on the clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively. It saves 10.86%, 16.44%, 23.33%, 60.24%, 80.29%, and 82.78% on the clock cycles of FLT-CTMI on 224-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively. It saves 40.86%, 41.43%, 41.8%, 39.53%, 38.34%, 37.60%, and 37.66% on the clock cycles of BOS on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively.

Modular inversions are used in the elliptic curve cryptography, for instance in the elliptic curve affine addition formulae. We compute 10^5 sets of an affine addition formula and anh affine double formula (ADD + DBL) with FLT-CTMI, BOS, BY, hdBY, and SICT-MI and evaluate the average clock cycles. The elliptic curves used in our experiments are NIST elliptic

curves, NIST-224, NIST-256, and NIST-384, with their base points. The results are presented in Table 7.11. ADD + DBL with SICT-MI outperformed all of the other algorithms except for FLT-CTMI on 256-bit. It saves 16.61%, 15.17%, and 17.67% on the clock cycles of that with hdBY on 224-, 256-, and 384-bit, respectively. It saves 11.59% and 18.29% on the clock cycles of that with FLT-CTMI on 224- and 384-bit, respectively. It saves 41.98%, 42.30%, and 42.48% on the clock cycles of that with BOS on 224-, 256-, and 384-bit, respectively.

Table 7.11: Comparison of average clock cycles for ADD+DBL.

	224 bits	256 bits	384 bits
FLT-CTMI	429307	320123	816605
BOS [Bos14]	654161	746821	1159968
BY [BY19]	532697	602731	972908
hdBY [PGrb21]	455161	507967	810430
SICT-MI	379539	430898	667217

Chapter 8

Conclusion and Future Works

8.1 Summary

Secure IoT devices require compact cryptosystems. ECCs that use shorter keys satisfy their requirements. However, with the development of SCAs, which are serious threats to the security of IoT devices, SCAs resistant and compact ECCs become the focus of research. Therefore, we focus on SCAs resistant and compact basic arithmetic operations of ECCs, ECSMs and modular inversions, in this study.

For ECSMs, in Chapter 4, we explored three notions (*Generality of k*, *Secure generality*, and *Executable coordinates*) of SCAs security of them and clarified what characteristics are needed for SCAs resistant ECSMs, firstly. Then, we investigated six scalar multiplication algorithms from these three notions, and thus, focused on Joye's regular 2-ary algorithms, which satisfy *Secure generality*. However affine coordinates with elliptic curve addition formulae are not *Executable coordinates* for them. To take advantages of the compact of affine addition formulae, we extended affine coordinates and improved Joye's regular 2-ary RL algorithm to make (extended) affine coordinates are *Executable coordinates* for it. Finally, SCAs resistant and compact RL ECSMs (Algorithms 23 and 24) are constructed.

In Chapter 5, we consider SCAs resistant ECSMs with better memory and computational efficiency. Considering LR ECSMs are expected to use less memory and ECSMs using affine addition formulae can be more efficient by reducing the number of modular inversions, we focus on SCAs resistant and compact LR ECSMs and modular inversions optimization. Firstly, for the modular inversions optimization, we studied eight affine combination-addition formulae of DA (Algrithms 25 and 26), DQ (Alg. 27), two-continuous-adds (Alg. 28), quadruple-add (Alg. 29), independent add and double (Alg. 30), independent two adds (Alg. 31), and independent adds and DQ (Alg. 32) in terms of memory and computational cost. Next, we improved Joye's regular 2-ary LR algorithm and proposed three SCAs resistant LR ECSMs (Algorithms 34–36). We proved that they satisfy *Secure generality* and (extended) affine coordinates are *Executable coordinates* for them. Note that both projective and Jacobian addition formulae with the same exceptional computations as the (extended) affine addition formulae can also be applied to our algorithms directly. We also revisited Algorithms 23–24 and extended the Alg. 24 to Alg. 33 to

optimize the number of modular inversions.

The key finding of Chapter 6 are as follows: Our EX-EF 4-ary RL algorithm (Alg. 33) with (extended) affine in combination with Alg. 32 is the most efficient when the ratio of I to M (or $r = \frac{I}{M}$) falls within the range of $\frac{11\ell-15}{\ell-11} < r \leq \frac{25\ell-41}{\ell+21}$ ($11.50 < r < 22.69$ for 224 bits, $11.43 < r < 22.96$ for 256 bits, and $11.28 < r < 23.60$ for 384 bits), assuming the computational costs of m_a and A are omitted and $S = M$. In our experiments, both the EX-EF 4-ary LR algorithm (Alg. 36) with (extended) affine in combination with Alg. 29 and the EX-EF 4-ary RL algorithm (Alg. 33) with (extended) affine in combination with Alg. 32 are highly efficient. EF 2-ary LR algorithm (Alg. 34) with (extended) affine and EF 2-ary LR algorithm with (extended) affine in combination with Alg. 25 use the least amount of memory for 12 field elements, reducing that of the Montgomery ladder with CA formulae and Joye's LR with CA formulae by 36.84%, that of 2-ary RL with (extended) affine by 14.29% and that of Joye's RL with CA formulae by 45.45%.

Our ECSMs can be applied to IoT devices and ECDSA signatures of blockchain satisfying a low memory requirement.

GCD or modular inversions are used in RSA key generation, affine addition formulae, and ECDSA. Therefore, we focus on them for SCAs resistant and efficient ECCs in Chapter 7. To establish constant-time algorithms that compute the GCD and modular inversions with short iterations and simple computations during one iteration, we defined the iteration formula in Definition 7.1. We proved that the iteration formula converges to $a_n = \gcd(a_0, b_0)$ and $b_n = 0$ with limited iterations in Theorem 7.1 and that the required number of iterations of the iteration formula is $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ in Theorem 7.2. Based on these, we proposed SICT-GCD and SICT-MI algorithms and compared their number of iterations and computations during one iteration with those of FLT-CTMI, BOS, BY, and hdBY. Finally, we experimentally evaluated the efficiency of the SICT-GCD and SICT-MI algorithms in comparison to the FLT-CTMI, BOS, BY, and hdBY algorithms. The results indicate that the SICT-GCD and SICT-MI algorithms are more efficient than hdBY. The number of iterations of hdBY is approximately 2.3 times the bit length of the larger input, and the number of iterations of SICT-GCD and SICT-MI algorithms is 2 times the bit length of the larger input. Thus, the longer the bit length of the larger input, the more iterations and time saved by our algorithms. SICT-GCD and SICT-MI achieved SCAs resistant and compact GCD and modular inversions.

8.2 Future Work

As the use of IoT devices continues to expand, there is a growing need for compact and efficient authentication methods. In light of this, the compact elliptic curve cryptosystem presented in this thesis must be adapted to the specific requirements of various IoT devices. Additionally, with the advent of quantum computing, current elliptic curve cryptography may be vulnerable to attacks, making it necessary to transition to isogeny-based elliptic curve cryptosystems. The techniques developed in this thesis will be crucial in this transition, as they can be applied to isogeny-based elliptic curve cryptosystems.

List of Publications

Journals

1. Yaoan Jin and Atsuko Miyaji. Compact elliptic curve scalar multiplication with a secure generality. *Journal of Information Processing*, volume 28, pages 464-472, 2020.

2. Yaoan Jin and Atsuko Miyaji. Secure and Compact Elliptic Curve Scalar Multiplication with Optimized Inversion. *The Computer Journal*, 2022.

International Conferences

1. Yaoan Jin, Chunhua Su, Na Ruan, and Weijia Jia. Privacy-preserving mining of association rules for horizontally distributed databases based on FP-tree. *The 12th international conference on information security practice and experience (ISPEC2016)*, Nov 16-18, Springer LNCS, Vol.10060, pp.300-314, 2016.

2. Yaoan Jin and Atsuko Miyaji. Secure and compact elliptic curve cryptosystems. *The 24th Australasian conference on information security and privacy (ACISP2019)*, July 3-5, Springer LNCS, Vol.11547, pp.639-650, 2019.

3. Yaoan Jin and Atsuko Miyaji. Secure and Compact Elliptic Curve LR Scalar Multiplication. *The 25th Australasian conference on information security and privacy (ACISP2020)*, Nov 30 – Dec 2, Springer LNCS, Vol.12248, pp.605-618, 2020.

4. Kiyofumi Tanaka, Atsuko Miyaji, and Yaoan Jin. Efficient FPGA Design of Exception-Free Generic Elliptic Curve Cryptosystems. *The 19th International Conference on Applied Cryptography and Network Security (ACNS 2021)*, June 21-24, Springer LNCS, Vol.12726, pp.393-414, 2021.

5. Yaoan Jin and Atsuko Miyaji. Short-Iteration Constant-Time GCD and Modular Inversion. *The 21th Smart Card Research and Advanced Application Conference (CARDIS 2022)*, Nov 7-9, Springer International Publishing, Cham, pp.82-99, 2023.

Domestic Conferences

1. Yaoan Jin and Atsuko Miyaji. Secure Elliptic Curve Scalar Multiplication without complete addition formulae. In *The 36th Symposium on Cryptography and Information Security (SCIS2019)*, Jan 22-25, 2019.

2. Yaoan Jin and Atsuko Miyaji. Secure and Compact Elliptic Curve Cryptosystems. In *The 85th CSEC*, 2019.

3. Yaoan Jin and Atsuko Miyaji. Secure and Compact Elliptic Curve Scalar Multiplication based on Affine. In *IEICE Japan Tech. Rep.*, vol. 119, no. 437, ICSS2019-105, pp. 321-333, 2020.

4. Yaoan Jin and Atsuko Miyaji. Secure and Compact Elliptic Curve LR Scalar Multiplication. In 電子情報学会信学技報, vol.120, no.112, ISEC2020-31, pp.111-118, 2020.

5. Yaoan Jin and Atsuko Miyaji. Efficient Modular Inversion Resisting Side Channel Attack. In *The 39th Symposium on Cryptography and Information Security (SCIS2022)*, Jan 18-21, 2022.

References

[AGTB18] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *Cryptology ePrint Archive*, 2018.

[AM11] Rahat Afreen and Suresh C. Mehrotra. A review on elliptic curve cryptography for embedded systems. *International Journal of Computer Science Information Technology (IJCSIT)*, 3(3):84–103, 2011.

[AMSSS17] Alejandro Cabrera Aldaya, Raudel Cuiman Márquez, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. Side-channel analysis of the modular inversion step in the RSA key generation algorithm. *International Journal of Circuit Theory and Applications*, 45(2):199–213, 2017.

[ASSS17] Alejandro Cabrera Aldaya, Alejandro J. Cabrera Sarmiento, and Santiago Sánchez-Solano. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering*, 7(4):273–285, 2017.

[BHH+14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security*, pages 157–175, Christ Church, Barbados, 3-7 March 2014. Springer, Berlin, Heidelberg.

[Bos14] Joppe W. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, 2014.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 340–398, 2019.

[CJ03] Mathieu Ciet and Marc Joye. (virtually) free randomization techniques for elliptic curve cryptography. In *International Conference on Information and Communications Security*, pages 348–359, Huhehaote, China, 10-13 October 2003. Springer, Berlin, Heidelberg.

[CKM21] Fabio Campos, Juliane Krämer, and Marcel Müller. Safe-error attacks on sike and csidh. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 104–125. Springer, 2021.

[CLM+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: an efficient post-quantum commutative group action. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 395–427. Springer, 2018.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *In International Conference on the Theory and Application of Cryptology and Information Security*, pages 51–65, Beijing, China, 18-22 October 1998. Springer, Berlin, Heidelberg.

[DFS19] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete (or how to evaluate the security of any leaking device), extended version. *Journal of Cryptology*, 32(4):1263–1297, 2019.

[dlFPS+21] Sadiel de la Fe, Han-Byeol Park, Bo-Yeon Sim, Dong-Guk Han, and Carles Ferrer. Profiling attack against RSA key generation based on a Euclidean algorithm. *Information*, 12(11):462, 2021.

[EL09] Nevine Ebeid and Rob Lambert. Securing the elliptic curve montgomery ladder against fault attacks. In *Proceedings of the 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 46–50, NW Washington, DC, United States, 6 September 2009. IEEE.

[ELM03] Kirsten Eisentrager, Kristin Lauter, and Peter L. Montgomery. Fast elliptic curve arithmetic and improved weil pairing evaluation. In *Cryptographers' Track at the RSA Conference*, pages 343–354, San Francisco, CA, USA, 13-17 April 2003. Springer, Berlin, Heidelberg.

[FGMN16] Pierre-Alain Fouque, Sylvain Guilley, Cédric Murdica, and David Naccache. Safe-errors on SPA protected implementations with the atomicity technique. In Jean-Jacques Quisquater Peter Y. A. Ryan, David Naccache, editor, *The New Codebreakers*, pages 479–493. Springer, Berlin, Heidelberg, 2016.

[GJM+11] Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on Weierstraß elliptic curves from Co-Z arithmetic. *Journal of Cryptographic Engineering*, 1(2):161, 2011.

[IT02] Tetsuya Izu and Tsuyoshi Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *International Workshop on Public Key Cryptography*, pages 280–296, Paris, France, 12-14 February 2002. Springer, Berlin, Heidelberg.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[Joy07] Marc Joye. Highly regular right-to-left algorithms for scalar multiplication. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 135–147, Vienna, Austria, 10-13 September 2007. Springer, Berlin, Heidelberg.

[Joy09] Marc Joye. Highly regular m-ary powering ladders. In *International Workshop on Selected Areas in Cryptography*, pages 350–363, Calgary, Alberta, Canada, 13-14 August 2009. Springer, Berlin, Heidelberg.

[JY02] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 291–302, Redwood Shores, CA, USA, 13-15 August 2002. Springer, Berlin, Heidelberg.

[Kal95] Burton S. Kaliski. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Design & Test of Computers*, 24(6):544–545, 2007.

[LM08] Patrick Longa and Ali Miri. Fast and flexible elliptic curve point arithmetic over prime fields. *IEEE Transactions on Computers*, 57(3):289–302, 2008.

[LMG+20] Xiaofang Li, Yurong Mei, Jing Gong, Feng Xiang, and Zhixin Sun. A blockchain privacy protection scheme based on ring signature. *IEEE Access*, 8:76765–76772, 2020.

[LN12] Duc-Phong Le and Binh P. Nguyen. Fast point quadrupling on elliptic curves. In *Proceedings of the Third Symposium on Information and Communication Technology*, pages 218–222, Ha Long, Vietnam, 23-24 August 2012. ACM.

[MLRB20] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. A compact and scalable hardware/software co-design of SIKE. Cryptology ePrint Archive, Paper 2020/040, 2020. https://eprint.iacr.org/2020/040.

[MM12] Atsuko Miyaji and Yiren Mo. How to enhance the security on the least significant bit. In *International Conference on Cryptology and Network Security*, pages 263–279, Darmstadt, Germany, 12-14 December 2012. Springer, Berlin, Heidelberg.

[MMM04] Hideyo Mamiya, Atsuko Miyaji, and Hiroaki Morimoto. Efficient countermeasures against RPA, DPA, and SPA. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 343–356, Cambridge, MA, USA, 11-13 August 2004. Springer, Berlin, Heidelberg.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[NSS17] Effy Raja Naru, Hemraj Saini, and Mukesh Sharma. A recent review on lightweight cryptography in IoT. In *International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 887–890, SCAD Institute of Technology, Palladam, 10-11 February 2017. IEEE.

[OSS07] Acıiçmez Onur, Gueron Shay, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, pages 185–203. Springer, 2007.

[PGrb21] Wuille Pieter, Maxwell Gregory, and roconnor blockstream. Safegcd-bounds. Github, 2021.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 403–428, Vienna, Austria, 8-12 May 2016. Springer, Berlin, Heidelberg.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SC21] Szymon Sarna and Robert Czerwinski. RSA and ECC universal, constant time modular inversion. In *AIP Conference Proceedings*, volume 2343, page 050004. AIP Publishing LLC, 2021.

[SRP02] Chari Suresh, Rao Josyula R., and Rohatgi Pankaj. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[Wal04] Colin D Walter. Simple power analysis of unified code for ecc double and add. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 191–204. Springer, 2004.

[Wro16] Michal Wronski. Faster point scalar multiplication on short Weierstrass elliptic curves over fp using twisted hessian curves over Fp2. *Journal of Telecommunications and Information Technology*, 3:98–102, 2016.

[XLC+18] Sen Xu, Xiangjun Lu, Aidong Chen, Haifeng Zhang, Haihua Gu, Dawu Gu, Kaiyu Zhang, Zheng Guo, and Junrong Liu. To construct high level secure communication system: CTMI is not enough. *China Communications*, 15(11):122–137, 2018.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.