HI-ECC: More Efficient Implementation Scheme of ECC on Standard RISC-V ISA

Jiankuo Dong, Wen Wu, Kaisheng Sun, Fu Xiao, Qingguan Gao, Fangyu Zheng, Jingqiang Lin.

***Abstract*—The development of service computing has led to an explosive growth in the number of devices, while RISC-V, leveraging its advantages in open standards, flexibility and customization, can provide more efficient, scalable, and customizable service architectures for service computing. Research on RISC-V based cryptography, especially public key cryptography with high computational complexity, can provide efficient cryptographic support for security authentication, signature generation, data encryption and so on. In this paper, based on the RISC-V 64-bit instruction set, we propose several methods to improve the performance of Curve25519/448 public key cryptography algorithm, abbreviated as HI-ECC. HI-ECC optimizes the implementation of Curve25519/448 cryptography from large integer representation, finite field, point arithmetic and scalar multiplication, in which the large integer operation optimizations can be extended to other elliptic curve public key cryptography schemes. Our HI-ECC also takes into account the side-channel protection security implementation, which ultimately meets the constant-time computing latency. On the same platform, the proposed HI-ECC demonstrates a 36.3% improvement over the state-of-the-art curve implementation on Curve25519, and a 32.3% enhancement on Curve448. Additionally, the proposed HI-ECC is versatile and applicable to other curves such as NIST P-256.**

***Index Terms*—Elliptic Curve Cryptography, Public Key Cryptography, RISC-V, Cryptography Engineering.**

I. Introduction

Service computing [2] has a bright future supporting the tremendous advances in emerging areas of computing such as cloud computing, big data, the Internet of Things (IoT) [3], edge computing, mobile computing, and beyond. With the growth of technological advancements and application demands, there has been an exponential increase in the number of devices utilized for service computing. Considering this scenario, RISC-V-based service computing emerges as a significant trend due to its advantages in open standards, flexibility, and customization, facilitating the realization of more efficient, scalable, and customizable service architectures. However, with the increasing proliferation of service computing devices and the expansion of their application scope, the importance of addressing security risks and vulnerabilities cannot be overlooked. The growing interconnectivity and complexity of these devices expose them to various potential threats, ranging from data breaches to malicious attacks. Cryptographic techniques provide security mechanisms [4] such as data encryption, identity authentication, and access control, ensuring the security and reliability of communication and data transmission.

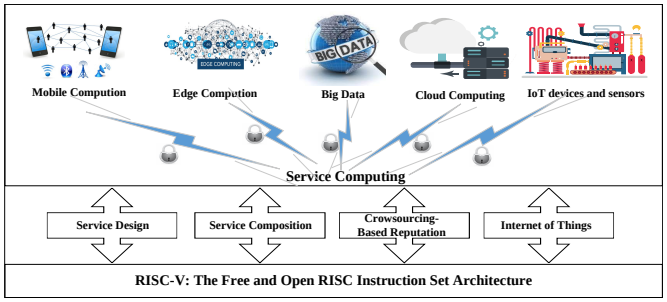


Fig. 1. RISC-V-based Service Computing

Public key cryptography, originating in 1976 [5], is fundamental to modern cryptographic practices, finding applications in various domains such as public key encryption, decryption, key negotiation [6], digital signatures [7], and secure multiparty computation [8]. While breaking the constraint of symmetric cryptography wherein encryption and decryption keys must match, public key cryptography introduces increased key length and computational complexity compared to symmetric methods [9]. The security of public key cryptography rests on mathematical problems, such as the RSA algorithm released in 1978, which relies on the challenge of large integer factorization. With advancements in computing, RSA keys must exceed 1024 bits [10] to mitigate exhaustive attacks. Alternatively, the elliptic curve cryptosystem [11], proposed in 1985, offers enhanced security, shorter key lengths, and greater flexibility by leveraging the elliptic curve discrete logarithm problem.

Curve25519, a Montgomery curve, stands out as one of the most high-speed elliptic curves to date. It was conceptualized and published by Professor Daniel J. Bernstein in 2006 [12]. Since its introduction, Curve25519 has garnered widespread attention in both academic and industrial spheres. Curve448, providing robust 224-bit security, gained favor from the Internet Research Task Force Crypto Forum Research Group (IRTF CFRG) and was chosen for inclusion in Transport Layer Security (TLS) standards, alongside Curve25519. Notably, Curve448 holds approval as an elliptic curve for use by the

A preliminary version of this paper appeared under the title"*V-Curve25519: Efficient Implementation of Curve25519 on RISC-V Architecture*" [1], in Proc. The 19th International Conference on Information Security and Cryptology, Inscrypt 2023, Nanjing, China, December 9-10, 2023.

J. Dong, W. Wu, K. Sun and F. Xiao are with School of Computer Science, Nanjing University of Posts and Telecommunications. Q. Gao is with School of Cyber Science and Engineering, Southeast University. F. Zheng is with School of Cryptology, University of Chinese Academy of Sciences. J. Lin is with School of Cyber Security, University of Science and Technology of China.

Corresponding author: ***Fu Xiao****, xiaof@njupt.edu.cn.*

US federal government, a status confirmed in FIPS 186-5. In January 2018, the IRTF released RFC 7748 [13], taking Curve25519/448 as the recommended curves of elliptic curve key negotiation protocol and naming them the X25519/448 key negotiation functions. Since the release of Curve25519/448, it has been widely used in various password libraries and security services. Starting from version 1.1.0, OpenSSL [14] includes support for the X25519 algorithm. Furthermore, in version 1.1.1, the X448 algorithm has been incorporated into the OpenSSL suite. In 2018, IRTF issued RFC 8446 [15], in which X25519 was used as the key negotiation algorithm of TLS protocol, and Ed25519 was added as the signature verification algorithm.

The X25519/448 protocols specifically provide key exchange functionalities, allowing two parties to securely establish a shared secret over an untrusted network. These curves have gained widespread adoption due to their excellent performance, simplicity, and resistance to various cryptographic attacks, making them well-suited for use in modern cryptographic applications and protocols. However, their computational complexity remains relatively high to a certain extent, especially when used in some low-performance embedded devices. Therefore, there is an urgent need to optimize the Curve25519/448 calculation process from bottom to top to improve the overall performance, so as to drive the rapid development of this cryptographic algorithm in the field of service computing.

RISC-V stands as a free and open-source ISA that originated at the University of California in 2010. RISC-V was founded as the RISC-V Foundation from 2015 and is incorporated today as RISC-V International Association [16]. Up to now, RISC-V can support 32-bit, 64 bit and even 128bit versions of secure microprocessors. Owing to its benefits of no licence fee, flexibility, scalability, free ISA that can be supported by the hardware, software and CPUs, RISC-V is distinctive—even revolutionary [17]. It possesses both concise and robust characteristics, capable of meeting the stringent requirements of embedded systems in terms of power consumption and cost, while also supporting high-performance computing demands. With the emergence of service computing, RISC-V has garnered attention for its flexibility and cost-free nature, rendering it an ideal platform for executing complex computational tasks.

A. Related works

Presently, researchers have successfully implemented Curve25519/448 curves on various high-performance processors or coprocessor platforms. Michael [18] accomplished the key agreement algorithm for X25519 on diverse embedded platforms. Armando [19] implemented the key agreement algorithm for ECDH utilizing the AVX2 instruction set. Koppermann [20] introduced a novel optimized architecture for X25519, specifically designed for low-latency applications in the realm of IoT devices. This architecture heavily exploits parallelisms in the Montgomery ladder algorithm. Hayato [21] implemented scalar multiplication of Curve25519 on the Cortex-M4 processor. Seo [22] and Anastasova [23] implemented scalar multiplication of Curve448 on the Cortex-M4 processor. Sasrich [24] implemented simultaneous scalar multiplication for both Curve25519 and Curve448 on FPGA. Gao et al. [1] implemented the Curve25519 scalar multiplication algorithm on the VisionFive @ 1.0 GHz platform, achieving a performance of 3,378 ops/s, which represents a 35% improvement over the state-of-the-art Curve25519 implementation on the same platform.

Stoffelen [25] introduced optimized assembly implementations for table-based AES, bitsliced AES, ChaCha, and the Keccak-$f[1600]$ permutation on the RV32I instruction set. Fritzmann [26] presented RISQ-V, an enhanced RISC-V architecture with tightly coupled accelerators designed to accelerate lattice-based PQC. RISQ-V efficiently reuses processor resources and minimizes memory accesses. Irmak [27] proposed a processor featuring an extended custom instruction for modular multiplication.

B. Our Contribution

In this paper, based on 64-bit RISC-V platform, the key negotiation algorithm functions X25519/448 based on Curve25519/448 are fully implemented, and the underlying finite field operations of Curve25519/448 scalar multiplication are meticulously designed, encompassing modulo addition, modulo subtraction, modulo multiplication, modulo square and modulo reduction, which significantly boost operational efficiency. Our High-performance Implementation of ECC (short for HI-ECC) has the advantages of high throughput, low delay, and high availability compared with existing works. The specific contributions of this paper are described in the following aspects:

- Firstly, based on RISC-V instruction set architecture, this paper fully realizes the key negotiation functions X25519/448 within the efficient elliptic curve cryptography Curve25519/448. Given the inadequacy of identity authentication performance in resource-constrained edge devices within service computing, we select the RISC-V embedded development board VisionFive as the encryption computing accelerator platform, in order to improve the computing efficiency of message encryption and identity authentication. Compared with other related work, the various optimization methods in this paper can significantly improve the performance of scalar multiplication of elliptic curves. Furthermore, the contributions of this paper extend beyond Curve25519/448, making it applicable to other elliptic curve cryptography algorithms and thereby yielding performance improvements.
- Secondly, utilizing the 64-bit RISC-V architecture core, this paper undertakes optimization of the finite field layer operation of the elliptic curve Curve25519/448 to fully exploit the word length advantage of a 64-bit processor. To our knowledge, this paper is the first to fully implement X25519/448 key negotiation algorithm based on the 64-bit RISC-V kernel, which fills in the gap of insufficient optimization of related algorithms on the 64-bit RISC-V processor. In addition, this paper uses

a constant computation delay scheme for the finite field large integer reduction method and uses the Montgomery Ladder algorithm with the same fixed delay to calculate the elliptic curve scalar multiplication, which makes the algorithm to some extent resistant to side-channel attacks.
- Finally, through the aforementioned optimization techniques, we leverage macro and loop unrolling techniques in our encoding process, determining the optimal loop unrolling factors. We delve from the bottom to the top layers into the scalar multiplication performance of Curve25519/448 on the RISC-V architecture, achieving the latest performance records for 64-bit RISC-V processors. Our experimental platform, VisionFive, demonstrates significant capabilities at 15W power consumption. It achieves 5224 scalar multiplications per second for curve25519 with a latency of 0.19 ms and 983 ops/s for curve448 with a latency of 1.02 ms. These results represent substantial improvements, being 1.36 times and 1.32 times faster than OpenSSL implementations on the same platform, respectively. Our optimization scheme offers a more efficient choice for security in IoT devices.

The rest of the paper is organized as follows. Section 2 provides an overview of essential concepts, including Curve25519/448, ECDH, RISC-V ISA, and details about our experimental platform, the VisionFive development board. Section 3 describes the proposed strategies for the key negotiation algorithm functions X25519/448 implementation. Section 4 performs our optimized implementation and compares it with related works. Section 5 concludes the paper and looks forward to future works.

II. Preliminary knowledge

In this chapter, first, the basic knowledge of elliptic curves Curve25519/448 is briefly described, then the Diffie-Hellman key negotiation algorithm and ECDH algorithm are introduced. Finally, the RISC-V instruction set architecture and the embedded development board StarFive VisionFive based on RISC-V 64-bit processor are introduced in detail.

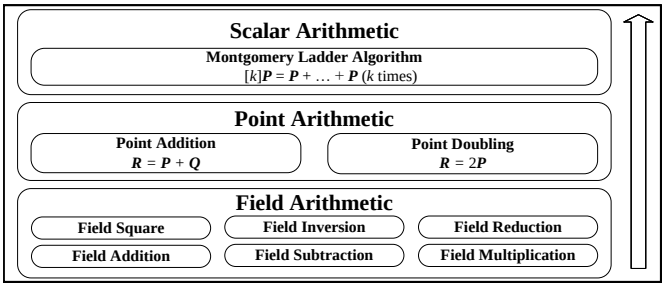


Fig. 2. The Architecture of ECC

A. Curve25519/448

As shown in Fig. 2, the architecture of ECC can be divided into finite field arithmetic layer, point arithmetic layer, and scalar multiplication arithmetic layer from bottom to top. Among them, the calculation of the upper layer needs to call the calculation of the lower layer to complete, so optimizing the calculation process at the arithmetic level of the lower finite field of the elliptic curve is the core of improving the efficiency of the elliptic curve cryptosystem.

Generally, the basic concept and operation principle of elliptic curve are introduced by taking Weierstrass curve as an example, because any elliptic curve can be written in the form of Weierstrass curve. In fact, elliptic curves also include many other types, such as Montgomery curve, twisted Edwards curve, and so on.

Weierstrass elliptic curve equation is as follows:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B \tag{1}$$

where $A, B \in \mathbb{F}_p$, $4A^3 + 27B^2 \neq 0$.

Montgomery elliptic curve equation is as follows:

$$E/\mathbb{F}_p : By^2 = x^3 + Ax^2 + x \tag{2}$$

where $A, B \in \mathbb{F}_p$, $B(A^2 - 4) \neq 0$.

Edwards elliptic curve equation is as follows:

$$E/\mathbb{F}_p : ax^2 + y^2 = 1 + dx^2y^2 \tag{3}$$

where $a, d \neq 0$, $a \neq d$.

Curve25519 was proposed by the famous cryptographer Daniel J. Bernstein [12]. It is an elliptic curve defined on the Montgomery curve and provides a security strength of 128 bits, where $p_1 = 2^{255} - 19$, $A_1 = 486662$, $B_1 = 1$. Similarly, as a Montgomery curve, Curve448 operates over a prime field of 448 bits in length, known as the Goldilocks prime field, offering a security strength of 224 bits, where $p_2 = 2^{448} - 2^{224} - 1$,$A_2 = 156326$, $B_2 = 1$.

$$Curve25519 : y^2 = x^3 + 486662x^2 + x \tag{4}$$

$$Curve448 : y^2 = x^3 + 156326x^2 + x \tag{5}$$

B. Diffie-Hellman Key Negotiation Algorithm and ECDH

Diffie-Hellman key negotiation algorithm was first proposed by Diffie and Hellman in an original paper in 1976 [28]. It is used by communication parties to negotiate a symmetric key in an insecure channel so that messages can be encrypted using the key in subsequent communication. It has been widely used in many security products.

ECDH applies elliptic curve to DH key negotiation and exchanges the keys of communication parties through the mathematical problem of elliptic curve discrete logarithm. The main calculation flow of ECDH is shown in Fig. 3:

1) Alice and Bob determine the curve type E, the finite field $\mathbb{F}_p$, and the base point G on the elliptic curve. n is the order of the base point G, that is, the minimum positive integer satisfying $nG = 0$;
2) Alice generates a random number $k_A \in (0, n-1)$ and calculates the scalar multiplication of elliptic curve $P_A = k_A G$;
3) Bob generates a random number $k_B \in (0, n-1)$ and calculates the scalar multiplication of elliptic curve $P_B = k_B G$;
4) Alice and Bob exchange P_A and P_B;

5) Alice and Bob respectively calculate $K_{share} = k_A P_B = k_B P_A$.

Since then, Alice and Bob have negotiated the key K_{share} only known to them and can communicate securely on the insecure channel through symmetric cryptography.

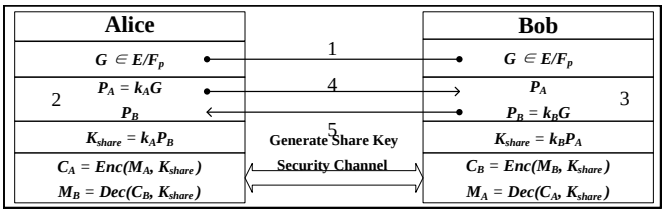


Fig. 3. The Diagram of ECDH

X25519/448 are elliptic curve key exchange algorithms based on the aforementioned curves. Compared with the traditional ECDH key negotiation protocol, the most remarkable feature of X25519/448 protocol is that it can calculate the x coordinate of kP only by relying on the x coordinate of point P on the elliptic curve. The idea of constructing ECDH key negotiation protocol using only x-coordinate came from the foundational paper published by Victor Miller in 1985 [29].

C. RISC-V and VisionFive

RISC-V instruction set architecture (ISA) was proposed by Andrew Waterman and others at the University of California, Berkeley in 2010. At present, its application has covered many fields such as IoT devices, desktop computers, high-performance computers, and so on. RISC-V is an open-source instruction set architecture, and its goal is to become Linux in the field of instruction set architecture. Andrew Waterman mentioned in his doctoral dissertation [30] that the existing instruction sets are all commercial instruction sets protected by patents and are not open, which not only restricts competition, but also curbs innovation, and brings certain obstacles to the development of the chip industry; On the other hand, the existing instruction sets have a long history and have undergone several twists and turns in the development process. Therefore, most of them carry a lot of historical burdens, resulting in the instruction sets being quite complex and not suitable for academic research. In this case, an open new instruction set architecture can not only be freely used by more people but also avoid many historical problems.

At the beginning of design, RISC-V instruction set is considered to be applicable to education and scientific research as well as to meet the application requirements in industrial scenarios. In order to achieve the above two purposes, RISC-V instruction set adopts the form of combining a basic instruction set and extension instruction sets, taking a basic instruction set as the core, and then adding different extension instruction sets according to functional requirements, just like adding plug-ins to an application program, which fully embodies the idea of modularization. RISC-V ISA has four basic instruction sets and six standard extension instruction sets, as shown in Table I.

TABLE I
COMPOSITION OF RISC-V ISA

Name	Numbers	Description
RV32I	47	Integer instruction, including arithmetic, branch, and memory access. 32 bits address space, 32 32-bit registers.
RV32E	47	The instruction is the same as RV32I, but the number of registers is changed to 16, which is used in embedded environment.
RV64I	59	Integer instruction, 64 bits addressing space, 32 64-bit registers.
RV128I	71	Integer instruction, 128 bits address space, 32 128-bit registers.
M	8	Contains 4 multiplication, 2 division, and 2 remainder operation instructions.
A	11	Contains atomic operation instructions, such as read-modify-write, compare-exchange, etc.
F	26	Contains single precision floating point instructions.
D	26	Contains double precision floating point instructions.
Q	26	Contains Quad Precision floating-point instructions.
C	46	Compressed instruction set, in which the instruction length is 16 bits, the main purpose is to reduce the code size.

VisionFive is a Linux single-board computer based on RISC-V ISA launched by the technology company StarFive. It can be widely used in edge computing, intelligent home appliances, intelligent monitoring, industrial robots, traffic management, intelligent logistics, wearable devices, network communications, and other fields. Visionfive embedded development board is equipped with 64-bit JH7100 CPU and 4GB LPDDR4 memory. The JH7100 processor is equipped with SiFive U74 dual-core and fully implements the RV64GC instruction set specification (where G is the initial letter of general and represents the specific combination of "IMAFD"). The operating frequency reaches 1.5GHz, which can provide strong computing capability for embedded intelligent devices such as edge computing, deep learning, and Internet of Things.

III. METHODOLOGY

This section describes our implementation method of the X25519/448 protocol based on RISC-V in detail. We first introduce the representation of large integers, including the concept and principle of radius limb. After that, we demonstrate how to calculate large integers at the finite field level under the representation method of radius limb, including addition, subtraction, multiplication, square etc. Subsequently, we elaborate on a detailed design of modular reduction strategies tailored for Curve25519/448, specifically optimized for the RISC-V platform. Finally, we carry out scalar multiplication of Curve25519/448 elliptic curve in single coordinate by Montgomery Ladder algorithm to fully implement X25519/448 key agreement protocol.

A. Representation of Elements

At present, a large number of works have done relevant research on the representation scheme of large integers. Fundamentally, the large number representation scheme needs to fully consider the characteristics of the target platform, so as to select the appropriate digital representation scheme.

1) Representation of elements in $\mathbb{F}_{p_1}$: For the 255-bit prime field elements used by curve25519, [5] initially proposed a scheme of Radix-$2^{25.5}$ to represent the field elements in the 32-bit architecture, which is widely spread and widely used in the industry. For the Radix-$2^{25.5}$ scheme, each 255-bit integer is composed of 10 limbs, including 5 length bits of 25 bits and the other 5 length bits of 26 bits. More formally, the field element k is represented by the following equation:

$$\begin{aligned} k = k_0 + 2^{26}k_1 + 2^{51}k_2 + 2^{77}k_3 + 2^{102}k_4 \\ + 2^{128}k_5 + 2^{153}k_6 + 2^{179}k_7 + 2^{204}k_8 + 2^{230}k_9 \end{aligned} \tag{6}$$

where $0 \leq k_{2i} < 2^{26}$ and $0 \leq k_{2i+1} < 2^{25}$ for $0 \leq i \leq 4$.

2) Representation of elements in $\mathbb{F}_{p_2}$: In the case of the 448-bit prime field elements utilized by curve448, [31] proposed a scheme of Radix-2^{29} to represent the field elements. Under the Radix-2^{29} scheme, each 448-bit integer is structured into 16 limbs, including 16 length bits of 29 bits. Similarly, the representation of the field element κ is expressed by the following equation:

$$\kappa = \sum_{i=0}^{15} \kappa_i \theta_i \tag{7}$$

where $0 \leq \kappa_i < 2^{29}$ and $\theta_i = 2^{29i}$ for $0 \leq i \leq 15$.

In the computation of a large integer represented by multiple limbs, when the lower limb surpasses the representable range, it becomes necessary to carry over to the higher limb. This carry is generally highly continuous (called carry propagation), which is not conducive to the operation efficiency of modern CPU. The Radix-$2^{25.5}$ representation scheme for Curve25519 has significant advantages, because, for a 26-bit number a, $19a$ is still within the range of 32-bit integers, so that the carry propagation can be effectively delayed until the end of the whole multiplication, thereby improving the efficiency of the algorithm to a certain extent. Similarly, the Radix-2^{29} representation scheme for curve448 can leverage the first 3 bits of each 32-bit integer to store the carry generated during the accumulation process of multiplication. This approach helps in avoiding carry propagation and contributes to the overall efficiency of the algorithm.

*3) **Radix-**2^{64} **Limb representation of elements**:* However, considering our RV64GC instruction set architecture and the processing capability of SiFive U74 core, this representation is not necessarily the best choice. Therefore, we have opted for Radix-2^{64} to represent the domain elements. For any k belonging to $\mathbb{F}p_1$ and κ belonging to $\mathbb{F}p_2$, each is composed of four 64-bit limbs. The representation equations are as follows:

$$Curve25519 : k = \sum_{i=0}^{3} k_i \theta_i \tag{8}$$

$$Curve448 : \kappa = \sum_{i=0}^{6} \kappa_i \theta_i \tag{9}$$

where $0 \leq k_i, \kappa_i < 2^{64}$ and $\theta_i = 2^{64i}$.

B. Implementation of Finite Field Arithmetic

As commonly acknowledged, the efficiency of the entire elliptic curve cryptosystem is heavily influenced by the underlying finite field arithmetic. Therefore, optimizing the performance of X25519/448 key agreement algorithm is to optimize the underlying finite field arithmetic. In this paper, finite field addition, finite field subtraction, finite field multiplication and finite field square are implemented based on Radix-2^{64}. Additionally, fast modulo reduction in the finite field is executed in accordance with the characteristics of modulo $p_1 = 2^{255} - 19$ and $p_2 = 2^{448} - 2^{224} - 1$, respectively.

In order to adapt to the Radix-2^{64} scheme used in this paper and improve the efficiency of the code, we have split up the calculation of finite field arithmetic and written the required calculation operations in the form of instructions in C language macro statements. The operation instructions involved in the experiment are shown in the Table II.

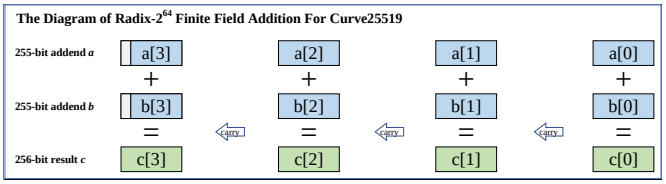
TABLE II
OPERATION INSTRUCTIONS

Instructions	Meanings
$ADD(a, b, c, s)$	$s = a + b$, and overflow value saved to c
$ADDC(a, b, c, s)$	$s = a + b + c$, and overflow value saved to c
$SUB(a, b, c, s)$	$s = a - b$, and underflow value saved to c
$SUBC(a, b, c, s)$	$s = a - b - c$, and overflow value saved to c
$MUL(a, b, h, l)$	$h = (a \times b)_{hi}$, $l = (a \times b)_{lo}$

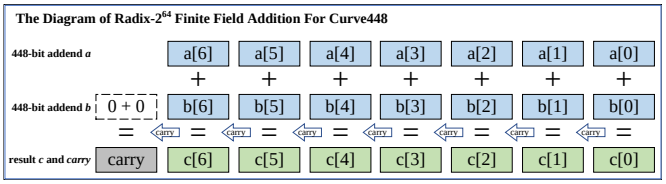
*1) **Finite Field Addition**:* For the implementation of finite field arithmetic operation on the bottom of an elliptic Curve25519, it is usually modified according to the specific large integer representation scheme to achieve a certain degree of optimization effect. For instance, work [32] uses 8 32-bit fixed points to represent a large integer of 255 bits, introducing redundant data of 1 bit. With the redundant bits of 1-bit, the input and output of the algorithm can be controlled at 256 bits. In the specific calculation process, only the large integer needs to be reduced to 256 bits, and only 256 bits can be reduced to $\mathbb{F}_p$, where $p = 2^{255} - 19$, when calculating the final scalar multiplication result. This technique can be used to cleverly simplify the rounds of reduction of large integers, thereby increasing the overall computational efficiency. However, this method requires an additional register to hold the overflow of 256-bit large integer addition results, which to some extent limits the overall performance. This paper uses 4 64-bit fixed-point numbers to represent a 255-bit large integer. Similar to the 8 32-bit fixed-point representation schemes, our representation scheme also has 1-bit redundancy, so we first tried the redundancy representation scheme in [32]. However, after many experiments, the performance of this scheme is not optimal on our RISC-V platform, so we simply chose the best implementation scheme.

Two 255-bit numbers are added in our finite-field addition algorithm, which means that the highest bit of the large integer addition input must be guaranteed to be zero to compute a 256-bit result. We don't need to immediately reduce the 256-bit calculation to 255-bit, but only when it is necessary to affect the correctness of the algorithm, in order to improve the overall

efficiency. The flow chart for finite field addition is shown in the Fig. 4, where the large integer addition $C = A + B$ utilizes one $ADD()$ macro instruction, two $ADDC()$ macro instructions and two additional simple additions.



(a) Finite Field Addition For Curve25519



(b) Finite Field Addition For Curve448

Fig. 4. Finite Field Addition

For Curve448, both the Radix-2^{64} representation scheme employed in this paper and the 32-bit scheme utilized in the work by [32] do not involve redundant bits. Despite the scheme using Radix-2^{29} representation, which can leverage redundant bits without requiring additional registers for carrying, our Radix-2^{64} representation method performs better, taking into account the computational characteristics of the RISC-V platform. It is noteworthy that the result of the addition calculation of large integers includes the large integer C and the $carry$. The value of the $carry$ is no greater than "1". The method of reducing the $carry$ will be elaborated in detail later. The large integer addition for Curve448 involves one $ADD()$ macro instruction, six $ADDC()$ macro instructions, and one *reduction*.

In addition, since RISC-V architecture does not provide the same functionality as CF in Program Status Word in X86 CPUs, programmers need to use software methods to determine whether data is overflowing or not. A common way to determine overflow is to compare the size of the operands and results. For example, as shown in Algorithm 1, in unsigned 64-bit integer addition $c = a + b$, if c is smaller than a or b instead, we can infer that a data overflow must have occurred. We can use a variable cy to hold the carry value, and when the carry occurs, it is set to "1", representing the 65th position of c. This makes it a carry operation in addition.

*2) **Finite Field Subtraction**:* The scheme for implementing subtraction in Curve25519 is similar to finite field addition. To avoid introducing additional registers to handle the "debits" that result from the calculation, the input for finite field subtraction is limited to 255-bit so that the overflow from the subtraction process can be recorded using the redundant space of up to 1 bit. In large integer subtraction $C = A - B$, if A is less than B, the highest bit of the result will also be set to "1", indicating an underflow. However, unlike the addition that does not require immediate reduction of 256-bit to 255-bit, the underflow handling and reduction of finite-field subtraction is not common to addition, multiplication, etc. Therefore, we have separately integrated the steps of reduction in the calculation of finite-field subtraction to make the result $C \in \mathbb{F}_{p_1}$, that is, to output the result of a 255-bit calculation. The flow chart for finite-field subtraction is shown in Fig. 5.

Algorithm 1: The carry generation method in addition

Input:
Two addends a and b;
Output:
The result c, and the carry cy generated by addition;
1: $c = a + b$
2: **if** $c < a$ **then**
3: $cy = 1$
4: **else**
5: $cy = 0$
6: **end if**

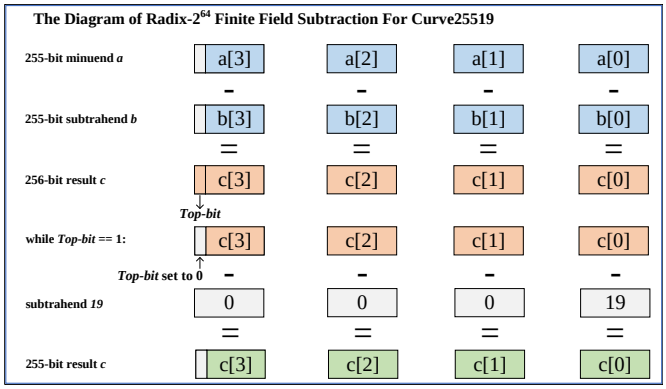


Fig. 5. Finite Field Subtraction For Curve25519

First, use the conventional method to calculate the 255-bit large integer subtraction $C = A - B$, and get the 256-bit calculation result. Then, a $while()$ loop is used to eliminate the possible "1" generated by the highest bit, that is, simply set it to zero, and then subtract 19 from the remaining large integer (because $2^{256} \equiv 19 \pmod{2^{255} - 19}$). Since the remaining large integers may be less than 19, the $while()$ loop will execute a maximum of 2 times.

However, the above scheme is not applicable to subtraction in Curve448 because it lacks redundant bits to store "borrow". Therefore, subtraction of large integers in Curve448 necessitates an additional register to store the borrow indicator. The implementation process of subtraction in Curve448 closely resembles addition, with the key distinction being the utilization of one $SUB()$ macro instruction, six $SUBC()$ macro instructions and one *reduction*

*3) **Finite Field Multiplication**:* Finite field multiplication is the most computationally intensive and time-consuming underlying operation besides inversion, which has a significant impact on the performance of any elliptic curve cryptography system. Therefore, optimizing the calculation of multiplication is essential.

For Curve25519, it operates within the finite field $\mathbb{F}_{p_1}$, where $p_1 = 2^{255} - 19$. As elucidated above, we use the Radix-2^{64} scheme, where a field element consists of four 64-bit numbers, to represent a large integer A, that is $A \in [0, 2^{256})$. Unlike finite field addition and subtraction, restricting the input of a finite field multiplication to 255-bit does not effectively reduce the use of additional variables, but rather results in some loss of efficiency due to additional reduction of the input. Therefore, in our multiplication calculation, the input can be 256-bit. For large integer multiplication $C = A \times B$ (where $A, B \in [0, 2^{256})$), there is $C \in [0, 2^{512})$.

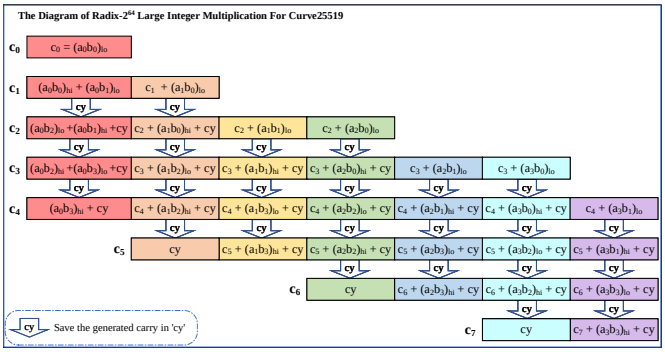


Fig. 6. Large Integer Multiplication For Curve25519

The basic unit that we use in the implementation of finite-field multiplication is the $MUL(a, b, h, l)$ macro instruction defined in TABLE II. Taking Curve25519 as an example, the specific implementation process is shown in the Fig. 6. As shown in the figure, we perform iteration calculations from left to right on a macroscopic level and update the calculation results. In other words, the rightmost grid of each line is the final calculation result of each limb. In addition, each column is calculated from top to bottom. In a nutshell, it is calculated in the order of rainbow colors. First, calculate the red column, then calculate the orange column, and so on, and finally iterate to calculate the final large integer multiplication result. The multiplication in Curve448 is the same as the multiplication in Curve25519, with the only difference being the number of operands, which will not be elaborated here.

*4) **Finite Field Square**:* For the calculation of the finite field square, since the two multiplier inputs are the same, that is, for the large integer multiplier $C = A \times B$ (where $A = B$), there is $A_i \times B_j = A_j \times B_i$. Therefore, there is a calculation method that can reduce the overall calculation steps by reusing intermediate results. To improve the overall performance of the algorithm, much literature [32], [33] has implemented the finite field square operation separately. The process of finite field square is divided into the following three steps:

- Firstly, when $i < j$, $A_i \times B_j$ (where $A = B$) is calculated, which is similar to the large integer multiplication mentioned above;
- Secondly, shift the above calculation result to the left by one bit. At this time, we have obtained the calculation result of $A_i \times B_j + A_j \times B_i$ $(i \neq j)$ (that is, twice as much as $A_i \times B_j$);
- Finally, for each i, compute $A_i \times A_i$, and add the results to the above results.

In the above calculation, the performance bottleneck lies in the left shift operation of the second step. Since the large integer representation scheme we use is Radix-2^{64} with 64-bit limb size, no redundant bits are reserved for each limb to use. Therefore, the $left\ shift$ operation needs to receive the highest bit data from the lower level branch and send its highest bit data to the lowest level of the higher level branch, which has a huge performance impact. We conducted experimental tests on the square calculation performance of Curve25519/448 using both multiplication and square methods, as shown in the Table III.

TABLE III
SQUARING IMPLEMENTATION COMPARISON

Curve Types	multiplication		Squaring	
	TP($10^6 ops/s$)	LAT(ns)	TP($10^6 ops/s$)	LAT(ns)
Curve25519	**17.857**	**56**	16.667	60
Curve448	**6.329**	**158**	5.525	180

In our experimental results for Curve25519/448, we observed that although the algorithm implemented for finite field squares can effectively reduce the number of simple multiplications, its overall performance is slightly weaker than that of using the multiplication algorithm directly for square calculations due to the influence of shift operations. Specifically, in our experiments, employing the multiplication algorithm directly for squaring calculations in Curve25519 improves the computational efficiency by approximately 7% compared to solely implementing the squaring algorithm. Conversely, for Curve448, utilizing the multiplication algorithm directly for squaring computations enhances the computational efficiency by roughly 15% when compared to implementing the squaring algorithm exclusively.

C. Fast Reduction

This paper presents the implementation of fast modular reduction algorithms for large integers, tailored for two distinct curves. Unique reduction schemes are provided for each of the two curves.

*1) **Fast Reduction For Curve25519**:* As gleaned from the previous section, the input of a large integer multiplication is two large 256-bit integers, which produce a 512-bit product output. For large integer multiplication $C = A \times B$ (where $A, B \in [0, 2^{256})$), there is $C \in [0, 2^{512})$. Further, C can be expressed as:

$$\begin{aligned} C &\equiv \sum_{i=0}^{7} C_i \times 2^{64i} \pmod{p_1} \\ &\equiv \sum_{i=4}^{7} C_i \times 2^{64i} + \sum_{i=0}^{3} C_i \times 2^{64i} \pmod{p_1} \end{aligned} \tag{10}$$

From $p_1 = 2^{255} - 19$, we can get: $2^{255} \equiv 19 \pmod{p_1}$ and $2^{256} \equiv 38 \pmod{p_1}$. Then, Equation 10 can be further reduced to:

$$\begin{aligned} C &\equiv \sum_{i=0}^{3} 38 \times C_{i+4} \times 2^{64i} + \sum_{i=0}^{3} C_i \times 2^{64i} \pmod{p_1} \\ &\equiv \sum_{i=0}^{3} (C_i + 38 \times C_{i+4}) \times 2^{64i} \pmod{p_1} \end{aligned} \tag{11}$$

It can be seen from Equation 11 that the 512-bit large integer multiplication result C is composed of 8 64-bit fixed-point numbers. For the large integer multiplication result C, we can take the digit with the higher 256 bits $(c_4c_5c_6c_7)$, multiply it by 38, and add it to the lower 256 bits of C $(c_0c_1c_2c_3)$. In this way, we get new calculation results, and it is obvious that the maximum length will not exceed 262 bits, of which the highest 6 bits are stored in a separate 32-bit variable, $carry$. At this time, we have made the first round of reduction for C, which is reduced from 512 bits to no more than 262 bits. Next, consider how to reduce 262 bits to 256 bits.

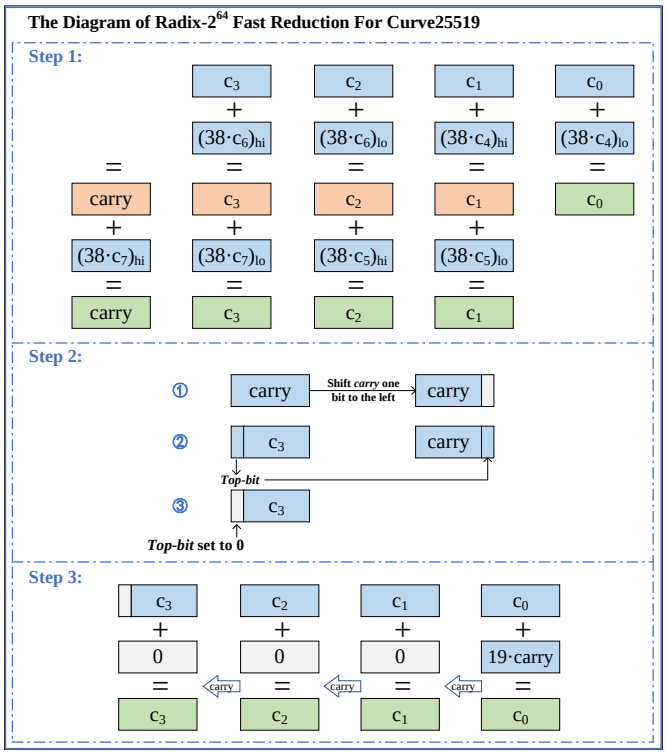


Fig. 7. Fast Reduction For Curve25519

The traditional computation method employs a $while$ loop instruction, repeatedly performing $(38 \times carry + C)$ additions until no new carry is generated. However, this approach has notable design flaws, as different inputs require varying loop iterations, resulting in different output delays and posing a risk of side-channel attacks. Therefore, we adopted an ingenious method, that is, we shifted the entire $carry$ to the left by one bit, and then moved the highest bit of the lower 256 bits part to the lowest bit of the $carry$. In this way, the lower 256 bits have 256 positions, but only 255 bits of information are saved. The value of carry cannot exceed 7 bits at most. According to equation $2^{255} \equiv 19 \pmod{p}$, carry can be multiplied by 19 and added to the lower 255 bits to obtain a calculation result not exceeding 256 bits. The reduction operation is specified as follows:

Algorithm 2: Fixed-duration further reduction process for Curve25519

Input:
256-bit large integer $C = \sum_{i=0}^{3} C_i \times 2^{64i}$, where $C_i \in [0, 2^{64})$, and $carry \in [0, 64)$;
Output:
256-bit large integer $C = \sum_{i=0}^{3} C_i \times 2^{64i}$, where $C_i \in [0, 2^{64})$;
1: $carry = carry \ll 1$
2: $carry+ = c_3 \gg 63$
3: $c_3\& = (1 \ll 63) - 1$
4: $carry* = 19$
5: **for** $i = 0$ *to* 2 **do**
6: $\quad C_i+ = carry$
7: $\quad carry = C_i < carry?1 : 0$
8: **end for**
9: $C_3+ = carry$

$$(carry \ll 1 | C_{256}) \times 19 + \widehat{C} \Rightarrow (carry_1 | C_1) \tag{12}$$

where C_{256} is the highest bit, $\widehat{C}$ is the lower 255 bits.

The computation result $(carry_1|C_1) \in [0, 2^{255} + 1462)$, which Indicates that the carry has been fully reduced. During the experimental process, the single-round carry reduction calculation method provided above can enhance the overall computational performance of the algorithm. However, this method is only applicable to the Curve25519 and cannot be applied to the Curve448. The individual steps for the single-round Curve25519 carry reduction calculation are outlined in Algorithm 2, and the overall fast reduction process is depicted in Fig. 7. Here, step 1 corresponds to Equation 11, step 2 corresponds to lines 1-3 in Algorithm 2, and step 3 corresponds to lines 4-9 in Algorithm 2.

*2) **Fast Reduction For Curve448**:* The aforementioned fast reduction method is not applicable to Curve448 as it lacks redundant bits. In comparison to Curve25519, the fast reduction process for Curve448 is more intricate.

For large integer multiplication, the input comprises two substantial 448-bit integers, yielding a 896-bit product output. In the case of large integer multiplication $C = A \times B$ (where $A, B \in [0, 2^{448})$), the result is within the range $C \in [0, 2^{896})$. Furthermore, C can be expressed as:

$$\begin{aligned} C &\equiv \sum_{i=0}^{13} C_i \times 2^{64i} \pmod{p_2} \\ &\equiv \sum_{i=7}^{13} C_i \times 2^{64i} + \sum_{i=0}^{6} C_i \times 2^{64i} \pmod{p_2} \end{aligned} \tag{13}$$

From $p_2 = 2^{448} - 2^{224} - 1$, we can get: $2^{448} \equiv 2^{224} +$

$1 \pmod{p_2}$. Then, Equation 13 can be further reduced to:

$$\begin{aligned} C &\equiv \sum_{i=0}^{6}[(2^{224}+1)C_{i+7}+C_i]\times 2^{64i} \pmod{p_2} \\ &\equiv \sum_{i=0}^{6}(C_i+C_{i+7})\times 2^{64i} + \sum_{i=3}^{6} C_{i+4}\times 2^{32}\times 2^{64i} \\ &+ \sum_{i=0}^{2} C_{i+11}\times 2^{32}\times 2^{64i} \\ &+ \sum_{i=0}^{2} C_{i+11}\times 2^{64(i+4)} \pmod{p_2} \end{aligned} \tag{14}$$

To prevent overflow, replace the terms multiplied by 2^{32} with & and $shift$ operations. Let $g(x) = (x\&mask_{32}) \ll 32$, $h(x) = (x \gg 32)$, where $mask_{32} = 2^{32}-1$. According to Equation 14, Algorithm 3 is carefully designed in this paper to reduce the result C from a length of 896 bits to 448 bits, while possibly generating additional carry.

Algorithm 3: Reduction process for Curve448

Input:
896-bit large integer $C = \sum_{i=0}^{13} C_i \times 2^{64i}$, where $C_i \in [0, 2^{64})$;
Output:
256-bit large integer $C = \sum_{i=0}^{6} C_i \times 2^{64i}$, where $C_i \in [0, 2^{64})$, and $carry$;

1: $carry = 0, u = 0$
2: $ADD(C_0, C_7, carry, C_0)$
3: **for** $i = 1$ *to* 6 **do**
4: $\quad ADDC(C_i, C_{i+7}, carry, C_i)$
5: **end for**
6: $u = carry$
7: $ADD(C_0, g(C_{11})|h(C_{10}), carry, C_0)$
8: $ADDC(C_1, g(C_{12})|h(C_{11}), carry, C_1)$
9: $ADDC(C_2, g(C_{13})|h(C_{12}), carry, C_2)$
10: $ADDC(C_3, g(C_7)|h(C_{13}), carry, C_3)$
11: $ADDC(C_4, g(C_8)|h(C_7), carry, C_4)$
12: $ADDC(C_5, g(C_9)|h(C_8), carry, C_5)$
13: $ADDC(C_6, g(C_{10})|h(C_9), carry, C_6)$
14: $u += carry$
15: $ADD(C_3, (C_{10} \gg 32) \ll 32, carry, C_3)$
16: **for** $i = 1$ *to* 3 **do**
17: $\quad ADDC(C_{i+3}, C_{i+10}, carry, C_{i+3})$
18: **end for**
19: $carry += u$

In the addition operations $ADD()$ and $ADDC()$ in Curve448, a carry of at most 1 bit length is generated. As indicated by Algorithm 3, the maximum value of the $carry$ does not exceed 3 (line 6.14.19), represented in binary as $(11)_2$. Consequently, the length of the $carry$ does not surpass 2 bits. Similarly, the further reduction of the $carry$ can be achieved through the following Equation 15.

$$\begin{aligned} C &\equiv \sum_{i=0}^{6} C_i \times 2^{64i} + carry \\ &+ (carry \ll 32)\times 2^{64\times 3} \pmod{p_2} \end{aligned} \tag{15}$$

Following one round of carry reduction, the computed result from the above formula may still generate a carry, denoted as $carry_0$ at this stage. The range of the product C_1 is as follows:

$$\begin{cases} [0, 2^{448}-1] & if \quad carry_0 = 0 \\ [3+3\times 2^{224}, 2^{448}+3\times 2^{224}+2] & if \quad carry_0 = 1 \end{cases} \tag{16}$$

Considering the above results, it can be deduced that C_1 falls within the range $[0, 2^{448}+3\times 2^{224}+2]$. At this stage, the newly generated $carry_1$ also belongs to the set $\{0,1\}$. Following the second round of carry neutralization, C_2 can be obtained as follows:

$$\begin{cases} [0, 2^{448}-1] & if \quad carry_1 = 0 \\ [3+3\times 2^{224}, 2^{226}+4] & if \quad carry_1 = 1 \end{cases} \tag{17}$$

The calculation indicates that through the two rounds of carry neutralization described above, the carry can be fully reduced. The complete reduction flow for Curve448 is shown in Fig. 8, which reduces the product C of large integer multiplication (squaring) of Curve448. Step 1 reduces the result from 896 bits to 448 bits with $carry$ according to Algorithm 3. Step 2 further completes the reduction of $carry$. According to the method of Fig. 8, two rounds of accumulation operation can completely reduce carry. Similarly, step 2 also supports reduction for large integer plus (minus) carryovers. In short, the implementation has the ability to resist timing attacks because the operation instruction time is determined independently of the input in the fast reduction operation.

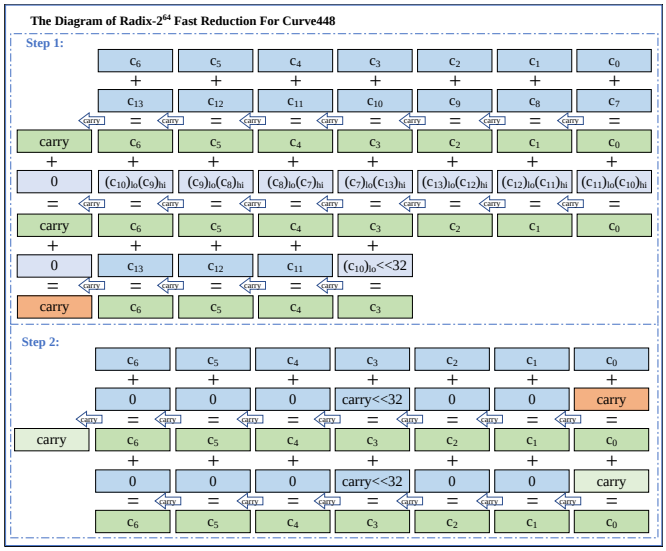


Fig. 8. Fast Reduction For Curve448

D. Implementation of Scalar Multiplication Arithmetic

The most traditional scalar multiplication scheme for variable base points of elliptic curves is the Double-add scheme, which receives cache-timing side channel attacks easily and is inefficient due to the input-related computation delay. The Montgomery Ladder scheme can well resist side-channel attacks due to the fixed latency of the computation. This paper uses Montgomery Ladder algorithm to realize scalar

multiplication of Curve25519/448 elliptic curve. The flow of the algorithm is shown in Algorithm 4, where $cswap()$ is a conditional exchange function and $swap$ is 1 to exchange the values of two variables. This function is completed using the scheme of literature [13], which also guarantees fixed calculation delay. In the Montgomery Ladder algorithm, there is a fixed data variable a_{24}, whose specific value in Curve25519/448 elliptic curve is:

$$Curve25519 : a_{24} = \frac{486662 - 2}{4} = 121665$$
$$Curve448 : a_{24} = \frac{156326 - 2}{4} = 39081$$

Since a large integer multiplication in the Curve25519/448 Montgomery Ladder algorithm process requires the fixed value $a_{24} + 1$, which does not exceed the maximum length of 17 bits, the full $256bits \times 256bits$ or $448bits \times 448bits$ multiplication is not required. In *step 22* of Algorithm 4, we separately implemented a simplified version of the modular multi-add algorithm for calculating the large integer multiplication and addition algorithm $C(256bits) = A(256bits) \times B(64bits) + D(256bits)$ or $C(448bits) = A(448bits) \times B(64bits) + D(448bits)$, which omits the corresponding part of $A \times (b_1, b_2, \cdots)$ calculation, and improves the overall efficiency of the algorithm. In addition, in order to improve the utilization of registers, reduce the swapping in and swapping out of memory, so as to make full use of the CPU pipeline and improve the computing efficiency, we fine tuned the order of some computing steps in the Montgomery Ladder algorithm, so that it can adapt to our RISC-V computing platform.

E. Macro and Loop Unrolling

Loop unrolling is a loop optimization technique aimed at enhancing loop performance by reducing branch occurrences and the number of loop maintenance instructions. Within loop unrolling, the body of the loop is repeated multiple times within the loop structure to improve performance. The number of repetitions is termed the loop unrolling factor, directly impacting the loop count, i.e., the original iteration count divided by the loop unrolling factor. Loop unrolling can significantly decrease the number of iterations, potentially eliminating them altogether. Moreover, loop unrolling increases the count of independently scheduled instructions, reducing instruction overhead and mitigating memory latency. In general, loop unrolling tends to lead to code verbosity. In this paper, the code within the loop body is implemented using macros. Through the utilization of macro parameters, the same macro definition is employed across different loops to accommodate the characteristic variations in each iteration. Leveraging macros for text replacement at compile-time, rather than utilizing function calls, eliminates the additional overhead associated with function calls, thereby further enhancing the algorithm's performance.

As shown in Algorithm 4, there exists a significant loop structure within Scalar Multiplication, which iterates 255 times for Curve25519 and 448 times for Curve448. In this paper,

Algorithm 4: The X25519/448 key negotiation function based on Montgomery Ladder algorithm

Input:
The scalar k, and x coordinate of random point P: u;
Output:
The x coordinate of scalar multiplication $k \cdot P$;

1: $x_1 = u, x_2 = 1, x_3 = u$
2: $z_2 = 0, z_3 = 1, swap = 1$
3: **for** $i = n - 1$ *to* 0 **do**
4: $k_i = (k >> 1)\&1$
5: $swap = swap \oplus k_i$
6: $(x_2, x_3) = \text{cswap}(swap, x_2, x_3)$
7: $(z_2, z_3) = \text{cswap}(swap, z_2, z_3)$
8: $swap = k_i$
9: $tmp1 = x_2 - z_2$
10: $x_2 = x_2 + z_2$
11: $tmp0 = x_3 - z_3$
12: $z_2 = x_3 + z_3$
13: $z_3 = tmp0 \times x_2$
14: $z_2 = tmp1 \times z_2$
15: $tmp0 = {tmp1}^2$
16: $tmp1 = {x_2}^2$
17: $x_3 = z_3 + z_2$
18: $z_2 = z_3 - z_2$
19: $x_2 = tmp1 \times tmp0$
20: $tmp1 = tmp1 - tmp0$
21: $z_2 = {z_2}^2$
22: $tmp0 = tmp1 \times (a_{24} + 1) + tmp0$
23: $x_2 = {x_2}^2$
24: $z_3 = x_1 \times z_2$
25: $z_2 = tmp1 \times tmp0$
26: **end for**
27: $(x_2, x_3) = \text{cswap}(swap, x_2, x_3)$
28: $(z_2, z_3) = \text{cswap}(swap, z_2, z_3)$
29: return $x_2 \times ({z_2}^{-1})$

the loop body code (lines 4-25) is written as a macro, denoted as $Loop(i)$, and used to replace the loop body. For instance, when the loop unrolling factor is set to 2, the original number of loop iterations of n can be reduced to $n/2$, and the number of checks and judgments also decreases to half of the original. The reduction in the number of loop iterations leads to a decrease in the number of independently scheduled instructions, thus achieving the goal of optimizing the loop. After conducting tests, we discovered that the optimal loop unrolling factor for Curve25519 is 4, while for Curve488, the best factor is 2. Increasing the loop unrolling factor further actually leads to a decline in performance, which can be attributed to the decreased cache utilization caused by code bloat, as well as the added complexity of the code that hinders compiler optimization. Taking Curve25519 as an example, the expansion with the unrolling factor set to 4 is shown in Table IV. In addition, this paper also adopts the same macro and loop unrolling techniques in multiplication calculations to enhance the performance of multiplication computation.

IV. Performance Evaluation

In this section, the experimental results of the optimized X25519/448 function on the VisionFive RISC-V development board are presented and compared with other work. The introduction to the VisionFive development board has been made earlier. The RISC-V utilized in our experiments operated

TABLE IV
LOOP UNROLLING FOR X25519

for $i = n - 1$ *to* 0 *with* $i- = 4$ **do**
  $Loop(i)$;
  $Loop(i-1)$;
  $Loop(i-2)$;
  $Loop(i-3)$;
end for

on the Debian operating system, compiled with gcc (Debian 11.3.0-3) toolchain. The CPU model employed was the SiFive U74 4-core 64-bit RV64GC @ 1.5GHz, with 4 GB LPDDR4 memory, and an approximate power consumption of 15W.

A. Experiment Results

For the 64-bit RISC-V instruction set architecture, we completed the performance optimization of the key agreement algorithm based on Curve25519/448 curve from the bottom finite field to the top scalar multiplication. The scheme in this paper can be used for the identity authentication and security computing of the Internet of Vehicles, making up for the shortcomings of existing algorithms in the adaptation of RISC-V 64-bit platform.

TABLE V
EXPERIMENT RESULTS

Operation Type	Curve25519 Cycles	Curve448 Cycles
Finite Field Addition	6	21
Finite Field Subtraction	13	22
Finite Field Multiplication	93	330
Finite Field Square	82	237
Finite Field Inversion	22,270	125,407
Finite Field MulConstAddNum	25	46
Scalar Multiplication	287,131	1,525,573

As shown in Table V, we show the number of $Cycles$ consumed in each operation of our implementation scheme. It can be seen from the table that the finite field subtraction operation consumes more instruction cycles than the addition operation. This is because the reduction of subtraction is different from addition and multiplication and needs to be implemented separately. The reason why the number of clock cycles required for the square calculation of curve25519/448 is less than that of multiplication has been explained in detail earlier. The inversion of finite fields and the Montgomery ladder algorithm are based on the addition, subtraction, multiplication, and square of finite fields. Therefore, the focus of subsequent research is still to continue to explore the optimal implementation of finite field multiplication.

B. Performance Comparison

Table VI summarize the X25519/448 unknown point scalar multiplication implemented based on the VisionFive development board for the Diffie-Hellman key negotiation protocol, and compare it with other platforms. Additionally, a reassessment of Gao's Curve25519 scalar multiplication algorithm [1] is conducted on the VisionFive @ 1.5 GHz platform, depicted in Fig. 9. Concurrently, we also benchmarked the performance of the latest version of OpenSSL on the same platform. This paper primarily evaluates the performance of our scheme from the following three aspects:

- **Cycles**: It represents the instruction cycles required to calculate X25519/448 function once;
- **Throughput** (ops/s): It indicates the number of X25519/448 key negotiation functions completed by the RSIC-V 64-bit CPU in a unit time;
- **Latency** (ms): It refers to the time required for X25519/448 on the RSIC-V 64-bit CPU platform from the calculation request to the end of the calculation.

As evident from the tables, the number of instruction cycles for elliptic curve scalar multiplication on most embedded platforms ranges from hundreds of thousands to millions, and the throughput is approximately tens of times per second. This limitation is attributed to both the computing platform's constrained performance and the inherent characteristics of the algorithm implementation. Even the fastest elliptic curve Four$\mathbb{Q}$ is claimed to have a computing speed of only about 350 times per second on ARM Cortex-M4F platform, which shows that the computing performance of related cryptographic algorithms on specific resource-constrained platforms is still insufficient to support massive computing requests.

FPGA, serving as semi-custom circuits, effectively offers a solution to the drawbacks of custom circuits and aptly addresses the limitations of programmable gate circuits. It finds extensive use in cryptographic operations. Sasdrich et al. [24] achieved Curve25519/448 point multiplication on the XC7Z7020 platform, with measured latencies of 0.75 ms and 1.4 ms, resulting in throughputs of 1339 ops/s and 623 ops/s, respectively. While their experimental results are comparatively lower than our experimental results on the RISC-V platform, FPGA's inherent advantage lies in its capability to parallelize computational processes, thereby necessitating a smaller number of cycles on this platform.

OpenSSL [14] is currently the most popular and widely used tool for SSL cryptographic libraries, and it is also the state-of-the-art implementation of cryptography algorithm on our test platform. It provides a generic, robust, and fully functional tool suite to support the implementation of the SSL/TLS protocol. The X25519 and X448 key negotiation algorithms are also incorporated into OpenSSL, and the performance of scalar multiplication can be assessed using the commands "$openssl\ speed\ ecdhx25519$" or "$openssl\ speed\ ecdhx448$". Following our tests on the VisionFive development board, the overall computing performance of the X25519/448 open-source crypto library in OpenSSL demonstrates its status as the implementation of the most advanced X25519/448 key agreement algorithm on the RISC-V 64-bit platform. It achieves this with the minimum number of single computing instruction cycles and delay, along with the maximum computing throughput.

In Table VI and Fig. 9, the experimental results of the scheme and the implementation of OpenSSL on our experimental platform, VisionFive, are detailed. The table illustrates that OpenSSL achieves a throughput of approximately 3834

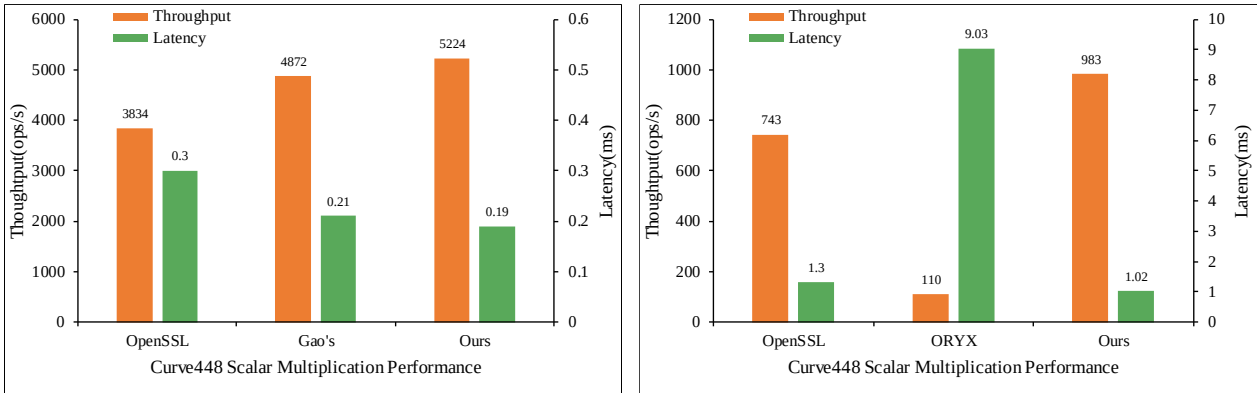


Fig. 9. Performance of Scalar Multiplication on VisionFive @ 1.5 GHz. (a) Performance of X25519; (b) Performance of X448. ("Gao's" denotes the experimental results of Gao et al. [1] on VisionFive @ 1.5 GHz; "ORYX" denotes the experimental results of ORYX [34] on VisionFive @ 1.5 GHz).

TABLE VI
CURVE25519/448 SCALAR MULTIPLICATION PERFORMANCE COMPARISON

	Platform	Curve	Cycles	TP (ops/s)	LAT (ms)
Liu et al. [35]	Cortex-M4F @ 400 MHz	Four$\mathbb{Q}$	880,642	350	2.79
Wei et al. [36]	Cortex-M0 @ 48 MHz	Four$\mathbb{Q}$	1,972,000	24	41.08
Nishinaga et al. [37]	Cortex-M0 @ 48 MHz	Curve25519	5,164,352	9	107.64
	Cortex-M0+ @ 72 MHz		4,209,866	17	58.48
Hayato et al. [21]	Cortex-M4 @ 48 MHz	Curve25519	907,240	52	18.90
	Cortex-M4 @ 72 MHz		1,003,707	71	13.94
	Cortex-M4 @ 84 MHz		894,391	93	10.65
Stefan [38]	Hifive1 @ 320 MHz	Curve25519	5,389,988	59	16.84
Seo et al. [22]	Cortex-M4 @ 24 MHz	Curve448	6,218,135	4	25.90
	Cortex-M4 @ 168 MHz		6,285,904	27	37.40
Anastasova et al. [23]	Cortex-M4 @ 24 MHz	Curve448	3,221,000	7	134.20
	Cortex-M4 @ 168 MHz		3,975,000	42	23.67
Niasar et al. [39]	XC7Z7020 @ 95 MHz	Curve448	133,254	713	1.40
ORYX [34]	VisionFive @ 1.5 GHz	Curve448	13,545,539	110	9.03
Sasdrich et al. [24]	XC7Z7020 @ 200 MHz	Curve25519	149,540	1,339	0.75
	XC7Z7020 @ 341 MHz	Curve448	547,728	623	1.30
OpenSSL [14]	VisionFive @ 1.5 GHz	Curve25519	498,000	3,834	0.30
		Curve448	1,952,000	743	1.30
Gao et al. [1]	VisionFive @ 1.0 GHz	Curve25519	295,967	3,378	0.29
	VisionFive @ 1.5 GHz		307,171	4,872	0.21
Ours	VisionFive @ 1.5 GHz	Curve25519	**287,131**	**5,224**	**0.19**
		Curve448	**1,525,573**	**983**	**1.02**

calculations per second for X25519 scalar multiplication and about 743 ops/s for X448 scalar multiplication. In comparison, the performance of the Curve25519/448 scalar multiplication implemented in this study surpasses that of OpenSSL. The throughput for Curve25519 is **5,224** ops/s, marking a **36.3%** improvement over OpenSSL, while the throughput for Curve448 is **983** ops/s, showing a **32.3%** enhancement compared to OpenSSL. Furthermore, our computing latency is smaller than that of OpenSSL implementation, and it can withstand the risk of side-channel attacks to some extent.

Furthermore, we re-evaluate the code of Gao's Curve25519 scalar multiplication algorithm [1] on the VisionFive @ 1.5 GHz platform, as shown in Fig. 9. The throughput obtained through retesting is 4,872 ops/s, with a latency of 0.21 ms. The improved X25519 scalar multiplication presented in this paper achieves a performance of 5,224 ops/s with a latency of 0.19 ms on the same platform, representing a 7.2% improvement compared to Gao's results. Simultaneously, we implement the Curve448 scalar multiplication algorithm from ORXY [34] on the RISC-V platform. The measured throughput is 110 ops/s with a delay of 9.03 ms, and the performance achieved in this paper is approximately 8.9 times higher.

V. Conclusion

Since the introduction of the RISC-V instruction set architecture with open attributes, it has become a hot breeze in the IoT world. In this study, leveraging the RISC-V 64-bit processor, we conducted extensive optimizations on the X25519/448 key negotiation functions. The achieved throughputs are respectively 1.36 and 1.32 times higher than those of the standard implementation in the OpenSSL cryptographic library, showcasing significant performance advantages. On the other hand, it also takes into account a certain anti-side channel attack capability. The proposed HI-ECC is versatile, applicable not only to Curve25519/448 but also to other curves, such as NIST P-{192,224,256,384,521}. In the future, we will focus on the computational performance of PQC on RISC-V architecture and continue to study the efficient implementation of related algorithms.

References

[1] Q. Gao, K. Sun, J. Dong, F. Zheng, J. Lin, Y. Ren, and Z. Liu, "V-curve25519: Efficient implementation of curve25519 on risc-v architecture," in *Information Security and Cryptology*, C. Ge and M. Yung, Eds. Singapore: Springer Nature Singapore, 2024, pp. 130–149.

[2] A. H. Hussein, "Internet of things (iot): Research challenges and future applications," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 6, 2019.

[3] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed *et al.*, "A service computing manifesto: the next 10 years," *Communications of the ACM*, vol. 60, no. 4, pp. 64–72, 2017.

[4] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan, "Internet of things (iot) security: Current status, challenges and prospective measures," in *2015 10th international conference for internet technology and secured transactions (ICITST)*. IEEE, 2015, pp. 336–341.

[5] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[6] B. Doyle, S. Bell, A. F. Smeaton, K. McCusker, and N. E. O'Connor, "Security considerations and key negotiation techniques for power constrained sensor networks," *The Computer Journal*, vol. 49, no. 4, pp. 443–453, 2006.

[7] C. F. Kerry and P. D. Gallagher, "Digital signature standard (dss)," *FIPS PUB*, pp. 186–4, 2013.

[8] O. Goldreich, "Secure multi-party computation," *Manuscript. Preliminary version*, vol. 78, p. 110, 1998.

[9] S. Chandra, S. Paira, S. S. Alam, and G. Sanyal, "A comparative survey of symmetric and asymmetric key cryptography," in *2014 international conference on electronics, communication and computational engineering (ICECCE)*. IEEE, 2014, pp. 83–93.

[10] Y. Suga, "Ssl/tls status survey in japan-transitioning against the renegotiation vulnerability and short rsa key length problem," in *2012 Seventh Asia Joint Conference on Information Security*. IEEE, 2012, pp. 17–24.

[11] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[12] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*. Springer, 2006, pp. 207–228.

[13] A. Langley and M. Hamburg, "Elliptic curves for security," *order*, vol. 500, p. 39081, 2016.

[14] O. S. Foundation, "OpenSSL Cryptography and SSL/TLS Toolkit," http://www.openssl.org/, 2016.

[15] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Tech. Rep., 2018.

[16] R.-V. International®. (2022) Risc-v international. [Online]. Available: https://riscv.org/

[17] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "A lightweight posit processing unit for risc-v processors in deep neural network applications," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1898–1908, 2022.

[18] M. Duell, B. Haase, G. Hinterwaelder, M. Hutter, C. Paar, A. H. Sanchez, and P. Schwabe, "High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Designs Codes & Cryptography*, vol. 77, no. 2-3, pp. 493–514, 2015.

[19] A. Faz-Hernández and J. López, "Fast implementation of curve25519 using avx2," in *Progress in Cryptology – LATINCRYPT 2015*, K. Lauter and F. Rodríguez-Henríquez, Eds. Cham: Springer International Publishing, 2015, pp. 329–345.

[20] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl, "Low-latency x25519 hardware implementation: breaking the 100 microseconds barrier," *Microprocessors and Microsystems*, vol. 52, no. jul., pp. 491–497, 2017.

[21] H. Fujii and D. F. Aranha, "Curve25519 for the cortex-m4 and beyond," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2017, pp. 109–127.

[22] H. Seo and R. Azarderakhsh, "Curve448 on 32-bit arm cortex-m4," in *Information Security and Cryptology–ICISC 2020: 23rd International Conference, Seoul, South Korea, December 2–4, 2020, Proceedings 23*. Springer, 2021, pp. 125–139.

[23] M. Anastasova, R. Azarderakhsh, M. M. Kermani, and L. Beshaj, "Time-efficient finite field microarchitecture design for curve448 and ed448 on cortex-m4," in *International Conference on Information Security and Cryptology*. Springer, 2022, pp. 292–314.

[24] P. Sasdrich and T. Güneysu, "Exploring rfc 7748 for hardware implementation: Curve25519 and curve448 with side-channel protection," *Journal of Hardware and Systems Security*, vol. 2, pp. 297–313, 2018.

[25] K. Stoffelen, "Efficient cryptography on the risc-v architecture," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2019, pp. 323–340.

[26] T. Fritzmann, G. Sigl, and J. Sepúlveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 239–280, 2020.

[27] Ö. F. Irmak and A. Yurdakul, "An embedded risc-v core with fast modular multiplication," *arXiv preprint arXiv:2009.14685*, 2020.

[28] W. Diffie and M. E. Hellman, "Multiuser cryptographic techniques," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, 1976, pp. 109–112.

[29] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 417–426.

[30] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[31] K. Nath and P. Sarkar, "Efficient 4-way vectorizations of the montgomery ladder," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–1, 2021.

[32] J. Dong, F. Zheng, J. Cheng, J. Lin, W. Pan, and Z. Wang, "Towards high-performance x25519/448 key agreement in general purpose gpus," in *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2018, pp. 1–9.

[33] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe, "High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers," *Designs, Codes and Cryptography*, vol. 77, no. 2-3, pp. 493–514, 2015.

[34] ORYX-EMBEDDED, "CycloneTCP, CycloneSSL, CycloneSSH, CycloneIPSEC, CycloneSTP, CycloneACME & CycloneCRYPTO," https://www.oryx-embedded.com/, 2023.

[35] Z. Liu, P. Longa, G. C. Pereira, O. Reparaz, and H. Seo, "Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 3, pp. 536–549, 2018.

[36] W. Zhang, D. Lin, H. Zhang, X. Zhou, and Y. Gao, "A lightweight fourq primitive on arm cortex-m0," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 699–704.

[37] T. Nishinaga and M. Mambo, "Implementation of μnacl on 32-bit arm cortex-m0," *IEICE TRANSACTIONS on Information and Systems*, vol. 99, no. 8, pp. 2056–2060, 2016.

[38] S. van den Berg, "Risc-v implementation of the nacl-library," Ph.D. dissertation, Master Thesis, 1 (1), 2020.

[39] M. B. Niasar, R. Azarderakhsh, and M. M. Kermani, "Optimized architectures for elliptic curve cryptography over curve448," *Cryptology ePrint Archive*, 2020.

Jiankuo Dong received the B.E. degree from the Xi'an Jiaotong University, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2014 and 2019, respectively. He is currently an Assistant Professor with School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography and applied cryptography.

Wen Wu received his B.E. degree from Nanjing University of Posts and Telecommunications in 2022, and currently pursuing an M.S. degree in the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering and high-performance computing.

Kaisheng Sun received his B.E. degree from Wanjiang University of Technology in 2021, and currently pursuing an M.S. degree in the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, applied cryptography, and high-performance computing.

Fu Xiao received the Ph.D. degree in computer science and technology from the Nanjing University of Science and Technology, Nanjing, China, in 2007. He is currently a Professor and a Ph.D. Supervisor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications. His research interest includes wireless sensor networks. Prof. Xiao is a member of the IEEE Computer Society and the Association for Computing Machinery.

Qingguan Gao received the bachelor and master degree from the Xi'an Jiaotong University. He is currently a PhD with School of Cyber Science and Engineering, Southeast University. His research interests include Cyber Range, Internet of Things security, and Internet of Vehicles security.

Fangyu Zheng received the B.E. degree from the University of Science and Technology of China, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2011 and 2016, respectively. He is currently an Assistant Professor with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences. His research interests include applied cryptography and high performance computing.

Jingqiang Lin , School of Cyber Security, University of Science and Technology of China, Hefei, China. Jingqiang Lin (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Chinese Academy of Sciences, in 2004 and 2009, respectively. He is a Full Professor with the School of Cyber Security, University of Science and Technology of China. His research interests include applied cryptography and system security.