SMCOS: Fast and Parallel Modular Multiplication on ARM NEON Architecture for ECC ⋆

Wenjie Wang[1,2], Wei Wang ✉[1,3], Jingqiang Lin ✉[4,5], Yu Fu[1,2], Lingjia Meng[1,2], and Qiongxiao Wang[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100089, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100089, China
[3] Data Assurance and Communication Security Research Center, CAS, Beijing 10089, China
[4] School of Cyber Security, University of Science and Technology of China, Hefei 230027, Anhui, China
[5] Bejing Institute, University of Science and Technology of China
{wangwenjie, wangwei}@iie.ac.cn, linjq@ustc.edu.cn

Abstract. Elliptic Curve Cryptography (ECC) is considered a more effective public-key cryptographic algorithm in some scenarios, because it uses shorter key sizes while providing a considerable level of security. Modular multiplication constitutes the "arithmetic foundation" of modern public-key cryptography such as ECC. In this paper, we propose the Cascade Operand Scanning for Specific Modulus (SMCOS) vectorization method to speed up the prime field multiplication of ECC on Single Instruction Multiple Data (SIMD) architecture. Two key features of our design sharply reduce the number of instructions. 1) SMCOS uses operands based on non-redundant representation to perform a "trimmed" Cascade Operand Scanning (COS) multiplication, which minimizes the cost of multiplication and other instructions. 2) One round of fast vector reduction is designed to replace the conventional Montgomery reduction, which consumes less instructions for reducing intermediate results of multiplication. Further more, we offer a general method for pipelining vector instructions on ARM NEON platforms. By this means, the prime field multiplication of ECC using the SMCOS method reaches an ever-fastest execution speed on 32-bit ARM NEON platforms. Detailed benchmark results show that the proposed SMCOS method performs modular multiplication of `NIST P192`, `Secp256k1`, and `Numsp256d1` within only 205, 310 and 306 clock cycles respectively, which are roughly 32% faster than the Multiplicand Reduction method, and about 47% faster than the Coarsely Integrated Cascade Operand Scanning method.

Keywords: Public-key cryptography, Vector instructions, Modular multiplication, SIMD, ECC, ARM NEON

⋆ ✉Wei Wang and ✉Jingqiang Lin are corresponding authors. This work was partially supported by Shandong Province Key Research & Development Plan/Major Science & Technology Innovation Project (Grant No. 2020CXGC010115).

1 Introduction

Effective implementation of public-key cryptographic algorithms on general-purpose computing devices facilitates the application of cryptography in communication security. As a crucial component of modern public-key cryptography, the Elliptic Curve Cryptography (ECC) based on the discrete logarithm problem has been widely used, because of its shorter key sizes than other cryptographic algorithms (e.g. DSA and RSA). Despite more than three decades of research efforts, ECC defined over general fields of large prime characteristic is still considered computation-intensive due to underlying arithmetic operations performed between large integers, especially when executed on embedded processors. Multi-precision modular multiplication is a performance-critical building block in ECC, which demands careful optimization to achieve acceptable performance [27].

In recent years, an increasing number of commodity processors were equipped with co-processors that provide vector instruction set extensions to perform single instruction multiple data (SIMD) operations. Advanced Vector Extension (AVX), the vector instruction set provided by Intel, is mostly used for application optimization on large server hosts and PCs. In terms of embedded platforms, due to the limitation of their computing capability, more and more ARM embedded processors (e.g. ARM Cortex-A, Cortex-R series) start to use NEON vector instructions to execute a wide variety of compute-intensive applications. For conventional cryptosystems, the parallel computing power provided by the SIMD co-processor can readily be used to optimize the implementation of public-key cryptographic algorithms such as RSA and ECC. In order to improve the performance of cryptographic algorithms, the research community has studied ways to reduce the latency of multi-precision modular multiplication through SIMD vectorization.

In these designs, one of the most often vectorized modular reduction techniques is the Montgomery algorithm [27, 3, 22, 26, 17, 31, 10, 25]. It was originally proposed in 1985 [19] and has been widely deployed in real-world applications. Montgomery modular multiplication, as a general modular multiplication design, has good execution efficiency and can be applied in multi-precision modular multiplication of cryptographic algorithms such as RSA and ECC. However, the Montgomery modular multiplication has the following two defects. One is that the instructions used by Montgomery reduction are usually expensive, which are roughly the same as the consumption of multiplication. On the other hand, a conditional subtraction could occur at the end of Montgomery modular multiplication in order to keep the result valid, which can be exploited in conventional timing-based side-channel attacks [25, 15, 29].

Besides, since the redundant representation suggested in [14] can handle carry propagation more easily, it is adopted by many vectorization solutions [2, 24, 5, 11, 10, 25, 16, 13]. The redundant representation allows several products of big numbers to be summed up, without causing an overflow inside the "container" (usually, a register) that holds the accumulation result. The cumbersome handling of the carry propagation can therefore be avoided [10]. However, the redundant representation introduces more multiplication instructions to compute

more partial products than the non-redundant representation. Also, when it is used for multiplicand or intermediate result reduction and carry propagation, the inconsistency of the size in bits of partial operands divided by redundant representation and the word size of the processors (32- or 64-bit) will cause additional overhead of instructions (e.g. *bic*, *shift* instruction) for handling reduction and carry.

In this paper, we propose an innovative design for ECC over the prime field $\mathbb{F}_p$, which uses non-redundant representation to implement a non-Montgomery form of vectorized modular multiplication, called Cascade Operand Scanning for Specific Modulus (SMCOS). Two key features of our design sharply reduce the number of instructions and pipeline stalls. 1) In a non-redundant representation, the multiplicands perform a "trimmed" Cascade Operand Scanning (COS) multiplication and obtain an intermediate result without carry propagation. COS vector multiplication was introduced in [27], which greatly eliminates Read-After-Write (RAW) dependencies in the instruction flow, and non-redundant representation reduces the number of multiplication instructions. When applied to SMCOS, the carry propagation at the end of COS is removed to avoid extra pipeline stalls due to sequential scalar operations in vector registers. 2) For the specific form of prime modulus in ECC, we introduce a fast vector reduction method in SMCOS to reduce the intermediate results of multiplication, instead of the general Montgomery reduction. The number of vector instructions consumed by this reduction is only about 12% $\sim$ 23% of the Montgomery reduction in [27] (see Sect. 4.2 for details). Furthermore, the SMCOS modular multiplication runs in constant time to resist certain types of side-channel attacks using timing and branch prediction.

On the Cortex-A9 platform, the SMCOS and two other fast vector modular multiplication methods for ARM NEON architecture, the Multiplicand Reduction (MR) [24] and the Coarsely Integrated Cascade Operand Scanning (CICOS) [27], are respectively integrated into the cryptographic algorithm library `OpenSSL 1.1.1k` [21], `libsecp256k1` [23] and `MSR ECCLib 2.0` [18]. After that, we make comprehensive comparisons of the execution time in terms of modular multiplication, point addition, point doubling, Elliptical Curve Diffie-Hellman (ECDH) for key exchange, Elliptic Curve Digital Signature Algorithm (ECDSA), etc. The detailed benchmark results indicate that SMCOS brings larger performance enhancements to all levels of ECC arithmetic. Taking ECDSA signature as an example, the signature performance of `NIST P192` curve based on SMCOS, is about 17% faster than the MR method, 22% faster than the CICOS method, and 58% faster than the native `OpenSSL` signature. And for `Secp256k1` curve, SMCOS's is roughly 10% faster than `libsecp256k1` optimized by manual assembly language before, and 26% faster than CICOS. Also for `Numsp256d1` curve, the signature performance using SMCOS is approximately 17% faster than CICOS and 25% faster than `MSR ECCLib` (see Sect. 5 for details).

The main contributions of our work are as follows.

- Firstly, a vector modular multiplication design based on specific modulus is proposed to fully exploit the computing power of SIMD co-processors for

ECC. To the best of our knowledge, this is the first non-redundant representation and non-Montgomery form of vector modular multiplication design in the prime field $\mathbb{F}_p$.

- Secondly, due to the specific modulus of the prime field for ECC, we design a single round of fast vector reduction method to reduce the intermediate results of multiplication.
- Thirdly, we investigate the timing of ARM SIMD integer instructions and provide a general method of pipelining on 32-bit ARM NEON platforms.
- Finally, thanks to highly optimized multiplication in $\mathbb{F}_p$, the performance of ECC protocols obtains great enhancements on 32-bit ARM processors with NEON.

The rest of the paper is organized as follows. Section 2 surveys the related work. The preliminaries about ARM NEON and the representation of prime field elements are presented in Section 3. Sections 4 describes the design and implementation of our SMCOS modular multiplication. In Section 5, performance results of the SMCOS method and ECC implementations are given and compared with other works and cryptographic algorithm libraries. We conclude in Section 6.

2 Related Work

The first practice and evaluation of cryptographic algorithm on ARM NEON architecture belonged to Bernstein and Schwabe in CHES'12 [2]. The authors showed that NEON supports elliptic curve cryptography at surprised high speeds, and summarized useful instructions for vectorization of cryptographic algorithms. In 2013, Câmaraand et al. employed NEON's `VMULL.P8` instruction to describe a novel vector implementation for 64-bit polynomial multiplication in ECC based on the binary field $\mathbb{F}_{2^m}$ [4]. [1, 30, 28, 9, 20] proposed accelerated implementations of applying NEON instructions to other cryptographic algorithms (e.g. AES, RSA, LWE, pairing-based and lattice-based cryptography, etc.). Despite recent research progress, for cryptographic algorithms, in particular public-key cryptography, the efficient implementation of multi-precision modular multiplication on the SIMD architecture is still an interesting and challenging topic.

The authors of [25] and [10] used Intel SSE and AVX2 vector instructions to implement Montgomery multiplication with redundant representation, and integrated them into RSA modular exponentiation. In SAC 2013, Bos et al. introduced a 2-way Montgomery modular multiplication, which uses non-redundant representation and splits the Montgomery modular multiplication into two parts: modular multiplication and reduction, being computed in parallel [3]. This is the first Montgomery modular multiplication parallel design with non-redundant representation, but its performance is compromised by the RAW dependencies in the instruction flow. Based on the work of Bos, Seo et al. proposed the Coarsely Integrated Cascade Operand Scanning (CICOS) method in ICISC 2014 [27]. This method eliminates the RAW dependencies of the 2-way Montgomery modular

multiplication in the carry propagation, thereby reducing the number of pipeline stalls and reaching record execution time.

In [24], the Multiplicand Reduction (MR) modular multiplication was introduced to implement NIST-recommended prime-field curves including `P192` and `P224`. The design adopts the redundant representation suggested in [14], and uses a kind of fast reduction instead of the Montgomery reduction to reduce multiplicands in advance. It is significantly faster than some schoolbook multiplication with intermediate reduction methods [21].

3 Preliminaries

3.1 ARM NEON architecture

The 32-bit RISC-based ARM architecture, which includes ARMv7, is the most popular in embedded devices. It features 13 general-purpose 32-bit registers (R0-R12), and additional three 32-bit registers which have special names and usage models: R13 for stack pointer, R14 for link register, as well as R15 for program counter. Its instruction sets support 32-bit operations or, in the case of Thumb and Thumb2, a mix of 16- and 32-bit operations [1].

Many ARM cores include NEON, a powerful 128-bit SIMD engine that comes with sixteen 128-bit registers (Q0-Q15) which can also be viewed as thirty-two 64-bit registers (D0-D31). The NEON instructions provide data processing and load/store operations, and are integrated into the ARM and Thumb instruction sets. NEON includes support for 128-, 16-, 8-, 4-, or 2-way SIMD operations using vectors of 1-, 8-, 16-, 32- and 64-bit integer elements respectively. The number of elements operated on is indicated by the specified register size. For example, `VADD.U8 Q0, Q1, Q2` indicates an addition operation on 8-bit integer elements stored in 128-bit Q registers. This means that the addition operation is on sixteen 8-bit lanes in parallel. Some instructions can have different size input and output registers. For example, `VMULL.U32 Q0, D2, D3` uses two pairs of 32-bit integers stored in two 64-bit D registers as inputs to generate a pair of 64-bit products and stores them in a 128-bit Q register. Similarly, there is a `VMLAL.U32` instruction that executes a `VMULL.U32` operation and adds the result to a 128-bit Q register (treated as two 64-bit integers). For more detailed information, refer to [12].

3.2 Representation of Prime Field Elements

The elements of $\mathbb{F}_p$ are usually represented by the integers in the range 0 to $p-1$ and the arithmetic operations remain as usual as in the integers except for the computation of a reduction modulo p at the end of each operation, which has the purpose of bringing the result within an original range. If p is a large integer of several hundreds or even thousands of bits, in order to store an $\mathbb{F}_p$ element in memory, an m-bit vector is needed, where m is the size of p in bits. However, the word size of prevailing processors is either $n = 32$ or $n = 64$ bits, which

in any case is shorter than the size of the large integer p. Therefore, multi-precision arithmetic must be implemented to handle integers larger than the word of processors [8]. At present, there are two popular designs for representing elements in $\mathbb{F}_p$, which are used in multi-precision arithmetic to divide an m-bit large number.

The *non-redundant* (*full-radix*) *representation* divides an $\mathbb{F}_p$ element into several parts with the word size of processors. In this way, an element can be stored by s words of n bits, i.e. $s = \lceil \frac{m}{n} \rceil$. The advantage of this representation is that its storage is compact, which usually means that fewer iterations are required to complete a multi-precision operation. However, one of the disadvantages of using this representation on an n-bit architecture is that some arithmetic operations impose a sequential evaluation of integer operations, for example, in the modular addition, the carry bits must be propagated from the least- to the most-significant digits. If non-redundant representation, there will be no extra space to store these carry bits, which limits the opportunities for calculating additions in parallel [7].

The second representation, *redundant* (*reduced-radix*) *representation*, divides an $\mathbb{F}_p$ element into s' shorter slices than the word size of processors, where $s' = \lceil \frac{m}{n'} \rceil$, $n' \in \mathbb{R}^+$ and $n' < n$. Because it relies on the selection of a real number $n' < n$, each word will have enough bits to store the carry bits produced by several modular additions. This feature can delay the carry propagation to the end and facilitate the implementation of parallelization. However, as mentioned above, compared to the non-redundant representation, it needs more iterations ($s' \geq s$) for completing a multi-precision operation, so more instructions are consumed.

4 Modular Multiplication for ECC Using SIMD Extensions

In this section, we firstly describe the design of the Cascade Operand Scanning for Specific Modulus (SMCOS) method for the prime field multiplication in ECC and its implementation details on the ARM NEON architecture. Then, we analyze the expected performance of our design, and compare it with the Multiplicand Reduction (MR) method in [24] and the Coarsely Integrated Cascade Operand Scanning (CICOS) method in [27]. Finally, we offer a general method of pipelining on 32-bit ARM NEON platforms.

4.1 Cascade Operand Scanning for Specific Modulus on SIMD

"Trimmed" COS. The COS Multiplication was first proposed in [27]. As a multiplication using non-redundant representation, it eliminates RAW dependencies in the instruction flow and has preferable efficiency. When it is used in SMCOS, we remove the carry propagation at the end of multiplication, which produces more pipeline stalls due to sequential scalar operations in vector registers.

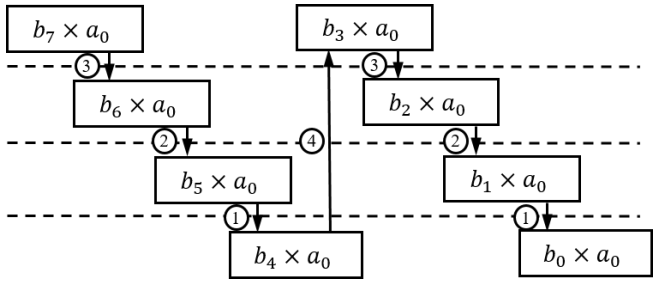

Fig. 1. Carry propagation in non-redundant representation. The lower bits are added to higher bits of lower intermediate results. The additions with same serial number are executed in parallel.

Taking the 32-bit word with 256-bit multiplication as an example, "trimmed" COS method is described in Algorithm 1. In the beginning, the algorithm conducts `VTRN` vector transpose instruction to re-organize and classify the operand $\bar{B}$ as groups $((b_7, b_3), (b_6, b_2), (b_5, b_1), (b_4, b_0))$ instead of the normal order $((b_7, b_6), (b_5, b_4), (b_3, b_2), (b_1, b_0))$. Next, in the first round, the products of (a_0, a_0) and elements in $((b_7, b_3), (b_6, b_2), (b_5, b_1), (b_4, b_0))$ are separately computed by `VMLAL` vector multiplication instruction, which supports 2-way multiplication in parallel. The partial product pairs are stored in $([L_7, L_3], [L_6, L_2], [L_5, L_1], [L_4, L_0])$, where each L_i is a 64-bit D register. Following which, the `VTRN` instruction is reused to separate the partial products into higher 32 bits ($63 \sim 32$) and lower 32 bits ($31 \sim 0$), generating eight pairs of 32-bit data stored in 16 D registers, $L_0 \sim L_7$ and $H_0 \sim H_7$. Finally, the lower bits are added to higher bits of lower intermediate results. For example, the lower 32 bits stored in $([L_7, L_3], [L_6, L_2], [L_5, L_1], L_4)$ are added to the corresponding higher 32 bits in $([H_6, H_2], [H_5, H_1], [H_4, H_0], H_3)$. By referring to Figure 1, this operation uses 3 vector addition `VADD` and 1 `ADD` instruction. After addition, the least significant word c_0 (lowest 32 bits of $\bar{B} \times a_0$) is obtained, and other more significant words are stored in H_0 to H_7.

In the next round, we need to perform $\bar{B} \times a_1$, because a_1 is higher than a_0, the products of (a_1, a_1) and $((b_7, b_3), (b_6, b_2), (b_5, b_1), (b_4, b_0))$ happen to be accumulated to intermediate results in $([H_7, H_3], [H_6, H_2], [H_5, H_1], [H_4, H_0])$ of the first round, and we can perform a new round of operations in the same way. This process is repeated with operands $(a_1 \sim a_7)$ by seven times more, we get the intermediate result $\bar{C}$ of $\bar{B} \times \bar{A}$. Its lower 256 bits are eight 32-bit values c_0 to c_7, which are the least significant words output at the end of each round. And higher 256-bit intermediate results are in L_0 to L_7 after the last round, 64-bit C_8 to C_{15}. The higher 32 bits of them are carry bits to upper intermediate results.

After that, the original COS multiplication will carry out the final carry propagation to align. Because it conducts sequential scalar operations directly in vector registers, the RAW dependencies incur more pipeline stalls. But in SMCOS, we keep the intermediate results of the multiplication to the next stage and straightforwardly reduce the results without pipeline stalls.

Algorithm 1 "Trimmed" COS. This arithmetic performs $\bar{B} \times \bar{A}$ and obtains the intermediate result $\bar{C}$. Note that $\bar{C}$ consists of two parts, c_i with a range of $0 \sim 2^{32} - 1$ and C_i with a range of $0 \sim 2 \times (2^{32} - 1)$.

Input: Two multiplicand $\bar{A}$ and $\bar{B}$ such that $\bar{A} = \sum_{i=0}^{7} a_i 2^{32i}, \bar{B} = \sum_{i=0}^{7} b_i 2^{32i}, 0 \leq a_i, b_i < 2^{32}$.

Output: Multiplication intermediate result $\bar{C} = \sum_{i=0}^{7} c_i 2^{32i} + \sum_{j=8}^{15} C_j 2^{32j}$.

```
B̄ ← VTRN(B̄)
Initialize L_i ← 0 for all i ∈ {0, 1, ..., 7}
for i = 0 to 7 do
    for j = 0 to 3 do
        [L_{j+4}, L_j] ← VMLAL([L_{j+4}, L_j], (a_i, a_i), (b_{j+4}, b_j))
    end for
    Initialize H_k ← 0 for all k ∈ {0, 1, ..., 7}
    for j = 0 to 3 do
        ([L_{j+4}, L_j], [H_{j+4}, H_j]) ← VTRN([L_{j+4}, L_j], [H_{j+4}, H_j])
    end for
    for j = 0 to 2 do
        [H_{j+4}, H_j] ← VADD([L_{j+5}, L_{j+1}], [H_{j+4}, H_j])
    end for
    H_3 ← ADD(L_4, H_3)
    c_i ← (L_0)_{0..31}
    Let L_j ← H_j for all j ∈ {0, 1, ..., 7}
end for
Let C_{i+8} ← L_i for all i ∈ {0, 1, ..., 7}
return C̄
```

Fast Reduction for Specific Modulus. Unlike most solutions [27, 3, 26, 17, 28] that use Montgomery reduction, we design a fast vector reduction method for the characteristic that most prime fields for ECC have specific modulus, and gain great performance advantages. We take `NIST P192` and `Secp256k1` as examples to illustrate different use cases of fast vector reduction on different curves, and offer our reduction method for modulo $P = 2^{256}–2^{32}–977$ in `Secp256k1`, see Algorithm 2.

In the reduction process, we will reduce the intermediate results of "trimmed" COS multiplication. For NIST-standard prime-field curves, NIST primes are special primes which are of the form $2^m \pm 2^n - ... - 1$. The smallest prime among NIST primes is $p_{192} = 2^{192}–2^{64}–1$, then any number larger than this prime can be reduced by using the relation $2^{192} \equiv 2^{64} + 1 (\mathrm{mod} p_{192})$. So for curves over NIST prime fields, we can use these relations to construct reduction for intermediate results of multiplication larger than NIST primes. Take `NIST P192` as an example, as shown in Figure 2, the intermediate results of 192-bit "trimmed" COS multiplication are separated into two groups. One group is the 32-bit c_0 to c_5 corresponding to $2^0, 2^{32}, ..., 2^{160}$ respectively, and they are less than p_{192}, so no reduction is required. We respectively store them in two 64-bit D registers on a Q register in pairs $((c_5, c_4), (c_3, c_2), (c_1, c_0))$. The second group

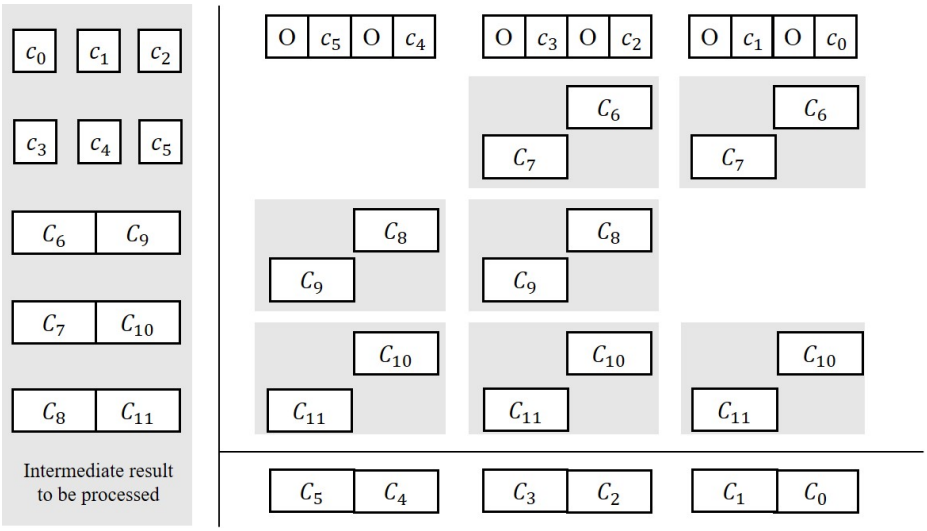

Fig. 2. One round of fast vector reduction for `NIST P192`. 32-bit S registers where "O" is located are cleared. The multiplication intermediate result to be processed is on the left, and the processing flow is on the right.

is the 64-bit intermediate results C_6 to C_{11} that are larger than p_{192}. They are items corresponding to $2^{192}, 2^{224}, ..., 2^{352}$. Using the above reduction relation, $C_6 \times 2^{192} \equiv C_6 \times 2^{64} + C_6 (\bmod p_{192})$. So, C_6 is reduced from 2^{192} to 2^{64} and 2^0, which correspond to the positions of c_2 and c_0 respectively.

In the same way, after reduction, the positions and times of C_7 to C_{11} can also be calculated, as shown in Figure 2. We find that for the intermediate results of the multiplication, the reduction can be further performed in an additive and parallel manner. By referring to Algorithm 1, the value range of C_6 to C_{11} is $0 \sim 2 \times (2^{32} - 1)$. Accumulating them several times with $c_0 \sim c_5$ will not result in an overflow of the 64-bit D register. We use the form of $([C_{11}, C_{10}], [C_9, C_8], [C_7, C_6])$ in pairs (the locations marked in Figure 2) and add them to the corresponding positions to complete all the reductions. Only 7 `VADD` vector additions are consumed, and we get six 64-bit reduction results, C_0 to C_5.

For the elliptic curves over non-NIST primes, take `Secp256k1` as an example. Although its modulo $P = 2^{256}–2^{32}–977$ is more irregular than NIST primes, the relation $2^{256} \equiv 2^{32} + 977 (\bmod P)$ still works. This relation results in some reduction items that may carry a multiplication factor, 977. As shown in Figure 3, c_0 to c_7 are items less than modulo P, we store them in four Q registers in pairs $((c_7, c_3), (c_6, c_2), (c_5, c_1), (c_4, c_0))$. For C_8 to C_{15}, we can also execute the reduction relation of modulo P to find out the positions of reduction items. But unlike NIST primes, we must multiply some items with the constant 977 to further transform their reduction to the method of NIST primes. Since 32-bit ARM NEON platforms do not provide 64-bit multiplication and vector mul-

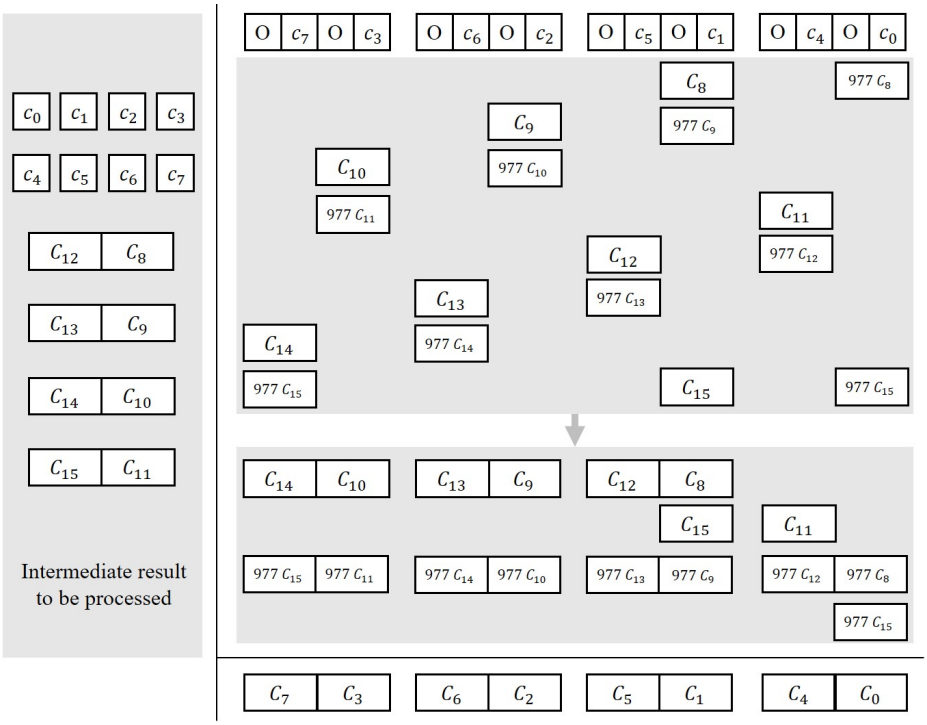

Fig. 3. One round of fast vector reduction for `Secp256k1`. 32-bit S registers where "O" is located are cleared. The multiplication intermediate result to be processed is on the left, and the processing flow is on the right.

tiplication instructions, we skillfully adopt vector shift (e.g. `VSHL`, `VSRA`) and vector subtraction `VSUB` to construct the multiplication, based on the relation of $977 = (2^{10} + 2^0)$–$(2^5 + 2^4)$. Fortunately, even if C_8 to C_{15} are multiplied by 977, they are far from beyond the range of the D register. Finally, after the multiplication with 977, we successfully conduct the reduction for modulo P in the similar manner with NIST primes, using 7 vector addition `VADD` and 3 `ADD` instructions.

For a more detailed description, see Algorithm 2, where `VSHL` is a vector shift left instruction, and `VSRA` is a vector shift right and accumulate instruction. In addition, for the third curve used in the experiment, `Numsp256d1`, the modulo P is 2^{256}–189. The reduction method in `Secp256k1` can be reused with a little transformation.

Final Alignment on Main Processor. SIMD co-processor is very effective in performing vector operations (e.g. parallel multiplication), but performs poorly for scalar operations like carry propagation and may pose more pipeline stalls [24]. Therefore, different from the previous vector modular multiplication designs [2, 27, 28], which deal directly with the final carry propagation and alignment

Algorithm 2 One round of fast vector reduction for Secp256k1. This arithmetic performs $\bar{C}' \equiv \bar{C}(\mathrm{mod}(2^{256}–2^{32}–977))$.

Input: Multiplication intermediate result $\bar{C}$ such that $\bar{C} = \sum_{i=0}^{7} c_i 2^{32i} + \sum_{j=8}^{15} C_j 2^{32j}, 0 \leq c_i < 2^{32}, 0 \leq C_j \leq 2 \times (2^{32} - 1)$.
Output: Reduction result $\bar{C}' = \sum_{k=0}^{7} C_k 2^{32k}$.

```
Initialize C_i ← c_i for all i ∈ {0, 1, ..., 7}
for i = 1 to 3 do
    [C_{i+4}, C_i] ← VADD([C_{i+4}, C_i], [C_{i+11}, C_{i+7}])
end for
C_4 ← ADD(C_4, C_11)
C_1 ← ADD(C_1, C_15)
for i = 0 to 3 do
    [C'_{i+12}, C'_{i+8}] ← VSHL([C_{i+12}, C_{i+8}], #5)
    [C''_{i+12}, C''_{i+8}] ← VSHL([C_{i+12}, C_{i+8}], #10)
    [C'_{i+12}, C'_{i+8}] ← VSRA([C'_{i+12}, C'_{i+8}], #1)
    [C''_{i+12}, C''_{i+8}] ← VSRA([C''_{i+12}, C''_{i+8}], #10)
    [C_{i+12}, C_{i+8}] ← VSUB([C''_{i+12}, C''_{i+8}], [C'_{i+12}, C'_{i+8}])
    [C_{i+4}, C_i] ← VADD([C_{i+4}, C_i], [C_{i+12}, C_{i+8}])
end for
C_0 ← ADD(C_0, C_15)
return C̄'
```

in vector registers, we design SMCOS as SIMD co-processor and ARM main processor working together. Multiplication and reduction are implemented using NEON vector instructions, but the final alignment is migrated to scalar registers. This change effectively breaks RAW dependencies in the instruction flow and reduces pipeline stalls. When carry bits are propagated to the most significant coefficient, no matter whether the digit (higher 32 bits of C_7 in Figure 3) larger than modulo P is 0, we will use reduction relations to perform a simple reduction and the second round of alignment, ensuring that SMCOS runs in constant time for resisting timing-based side-channel attacks.

4.2 Performance Analysis

In this section, we analyze the performance of our Cascade Operand Scanning for Specific Modulus (SMCOS) method, and compare it with the Multiplicand Reduction (MR) method in [24] and the Coarsely Integrated Cascade Operand Scanning (CICOS) method in [27].

For the clock cycle of instructions on the ARM NEON architecture, we denote 2-cycle instructions (e.g. `VMULL`, `VMLAL`, etc.) as X, and 1-cycle instructions (e.g. `VADD`, `ADD`, `VTRN`, etc.) as Y. For the modular multiplication in `NIST P192`, in the process of multiplication, the SMCOS and CICOS methods using non-redundant representation need to be executed 6 rounds. In each round they mainly conduct 3 `VMLAL` (`VMULL`), 3 `VTRN`, 2 `VADD`, and 1 `ADD` instructions, the instructions consumed in each round are equal to $3X + 6Y$. Therefore, for the

Table 1. Comparison of instructions for modular multiplication. X represents a 2-cycle instruction and Y represents a 1-cycle instruction.

Elliptic Curve	Stage	MR[24]*	CICOS[27]	Our SMCOS
NIST P192	Multiplication	$32X$	$18X + 36Y$	$18X + 36Y$
	Reduction	$35Y$	$18X + 36Y$	$7Y$
	Final Alignment	$48Y$	$12Y$	$12Y$
	Total	$32X + 83Y$	$36X + 84Y$	$18X + 55Y$
Secp256k1	Multiplication	-	$32X + 64Y$	$32X + 64Y$
	Reduction	-	$32X + 64Y$	$30Y$
	Final Alignment	-	$16Y$	$16Y$
	Total	-	$64X + 144Y$	$32X + 110Y$
Numsp256d1	Multiplication	-	$32X + 64Y$	$32X + 64Y$
	Reduction	-	$32X + 64Y$	$24Y$
	Final Alignment	-	$16Y$	$16Y$
	Total	-	$64X + 144Y$	$32X + 104Y$

*The MR method is not applicable to `Secp256k1` and `Numsp256d1`.

SMCOS and CICOS methods, their total instructions in the multiplication process are approximately $18X + 36Y$. The MR with redundant representation is 8 rounds in total, and each round only conducts `VMULL` (`VMLAL`) four times, which is equal to $4X$, and its total execution instructions are about $32X$.

In the reduction stage, the SMCOS method only requires one round, seven `VADD` instructions, so the total number of instructions used is $7Y$. Each round of MR mainly requires 2 `VEXT`, 1 `BIC`, and 2 `ADD` instructions, 7 rounds in total, and the instructions can be represented as $35Y$. As for CICOS, its reduction and multiplication are all completed by a COS multiplication, so the instructions used for reduction are also about $18X + 36Y$. In the final alignment, the instructions consumed by the SMCOS and CICOS methods with non-redundant representation are mainly additions, and each alignment needs carry only six times, and the total instructions are about $12Y$. Compared with them, the MR using redundant representation also requires a *shift* and a *bic* instruction to complete carry, and each alignment executes 8 times, so the total instructions are roughly $48Y$.

Based on the same standard, we count the instructions of the SMCOS and CICOS methods at each stage, when they are applied to `Secp256k1` and `Numsp256d1`. As shown in Table 1, both 2-cycle instruction (X) and 1-cycle instruction (Y) used by SMCOS are significantly reduced compared to MR and CICOS methods. According to Table 1, it can be further estimated that clock cycles of the instructions conducted by the SMCOS method are reduced by about 38% compared with MR, and about 36% $\sim$ 42% compared with CICOS. In particular, for the vector modular multiplication designs implemented by NEON, the main 2-cycle instructions used are `VMULL` and `VMLAL` vector multiplication instructions. For the two instructions with larger execution cycles, the SMCOS method greatly reduces the frequency of their use, which is mainly reflected in the following two

aspects. 1) In the multiplication stage, we use a non-redundant representation, which reduces the number of partial products compared to the MR method with redundant representation. Taking `NIST P192` as an example, MR uses a radix-2^{24} representation (i.e. 24 bits per word) for 192-bit operands, the total number of partial products is $8 \times 8 = 64$, which requires 32 vector multiplication instructions. On the other hand, SMCOS uses non-redundant representation based on a radix of 2^{32}, and reduces the number of partial products to $6 \times 6 = 36$, only 18 vector multiplication instructions. Besides, there are also fewer other instructions for reduction and carry propagation, because the size in bits of operands separated in a non-redundant way is the same as the word size of ARM processors. 2) In the reduction stage, our choice is not the Montgomery reduction adopted by CICOS, because it consumes the same instructions as the multiplication stage. A fast vector reduction design is used by SMCOS, so that SMCOS does not need to use any multiplication and only requires some instructions (e.g. addition, shift, and subtraction) with smaller clock cycles to complete the reduction.

4.3 Making SMCOS Fully Pipelined

Data dependencies in the instruction flow may cause pipeline stalls during the execution of vector instructions. If an instruction about to be executed has to wait for the operands written by the previous instruction for several cycles, in the meantime no other instructions enter the pipeline, the cycles of SIMD co-processors will be wasted and the performance will be compromised [31]. This kind of data dependencies between instructions are called Read-After-Write (RAW) dependencies, and the purpose of pipelining is to reduce or avoid RAW dependencies. Due to a large number of pipeline stalls, the 2-way Montgomery modular multiplication in [3] even obtains lower performance than scalar methods, when they are all implemented on ARM. Therefore, in order to maximize the performance of SMCOS, we need to perform sophisticated pipelining. And we investigate the clock cycles and delay of the instructions used by SMCOS. The advanced SIMD integer instruction timing on ARM Cortex-A9 platforms is provided in [6], as shown in Table 2.

We conduct sophisticated pipelining in each stage of the SMCOS implementation. During the execution of SMCOS, every vector instruction will be performed in terms of the sequence in the assembly code, so we manually adjust the instruction sequence to avoid pipeline stalls. Take the construction of vector multiplication with 977 in Algorithm 2 as an example, the original assembly code is ASM Code 1 in Figure 4. According to the timing of vector instructions in Table 2, the manually adjusted assembly code is ASM Code 2 in Figure 4.

As described in Algorithm 2, ASM Code 1 uses 2 `VSHL`, 2 `VSRA`, and 1 `VSUB` instructions to construct a vector multiplication with constant 977 on four pairs of 64-bit data stored in Q8 to Q11. The processing code of Q8 is in lines 1 to 5 of ASM Code 1, and the codes of Q9 to Q11 can be deduced by analogy. There are a lot of RAW dependencies in the original assembly code. Even though the execution cycle of the 20 instructions in ASM Code 1 is only 20 clock cycles in total, in fact, according to Table 2, due to pipeline stalls caused by the

Table 2. Advanced SIMD integer instruction timing on ARM Cortex-A9[6]

Instruction	Description	Issue Cycles*	Available Result**
VADD	Vector Addition	**1**	**3**
VDUP	Vector Duplication	**1**	**2**
VMOV	Vector Move	**1**	**3**
VSWP	Vector Swap	**1**	**2**
VSUB	Vector Subtraction	**1**	**3**
VEXT	Vector Extraction and Concatenate	**1**	**2**
VTRN	Vector Traspose	**1**	**2**
VSHL	Vector Shift Left	**1**	**3**
VSRA	Vector Shift Right with Addition	**1**	**4**
VMULL	Vector Multiplication	**2**	**7**
VMLAL	Vector Multiplication with Addition	**2**	**7**

*This is the number of issue cycles the particular instruction consumes.
**The Result field indicates the execution cycle when the result is ready.

instruction dependencies, it takes 50 cycles to complete execution and get all the results. This is absurd but true. In order to reduce pipeline stalls, we insert several independent instructions into any two data-dependent instructions to break these dependencies, so that the pipeline can be filled with new instructions again and fully utilized while an instruction is waiting for the result of the previous instruction. By referring to Table 2, we perform pipelining to ASM Code 1, the adjusted ASM Code 2 only takes 22 clock cycles to get all results, which is 44% of ASM Code 1.

5 Results

In this section, we conduct the experiments to evaluate our SMCOS method and ECC implementations on the 32-bit ARM Cortex-A9 processor and compare our results with related work and several ECC algorithm libraries, in terms of modular multiplication, point addition, point doubling, ECDH, and ECDSA.

5.1 Target Platforms

The ARM Cortex-A series are full implementations of the ARMv7, v8 architecture including NEON engine. The Cortex-A processors provide a series of application scenarios for devices using operating systems such as Linux or Android. These devices are used in various applications, from low-cost handheld devices to smartphones, tablets, set-top boxes, and corporate network devices. Among the Cortex-A series processors, we choose the Cortex-A9 on 32-bit ARMv7 architecture as the experimental platform, which is consistent with the previous implementations [27, 28, 3, 17, 16, 1]. The Cortex-A9 processor is widely used in several devices including iPad 2, iPhone4S, Galaxy S2, Galaxy S3, Galaxy Note 2, Kindle Fire, and NVIDIA Tegra T30. At the same time, the NEON instructions on its ARMv7 architecture are compatible with ARMv8.

```
vshl.u64 q12, q8,  #5
vsra.u64 q12, q12, #1
vshl.u64 q8,  q8,  #10
vsra.u64 q8,  q8,  #10
vsub.u64 q8,  q8,  q12
vshl.u64 q13, q9,  #5
vsra.u64 q13, q13, #1
vshl.u64 q9,  q9,  #10
vsra.u64 q9,  q9,  #10
vsub.u64 q9,  q9,  q13
vshl.u64 q14, q10, #5
vsra.u64 q14, q14, #1
vshl.u64 q10, q10, #10
vsra.u64 q10, q10, #10
vsub.u64 q10, q10, q14
vshl.u64 q15, q11, #5
vsra.u64 q15, q15, #1
vshl.u64 q11, q11, #10
vsra.u64 q11, q11, #10
vsub.u64 q11, q11, q15
```

(a) ASM Code 1 (Original)

```
vshl.u64 q12, q8,  #5
vshl.u64 q8,  q8,  #10
vshl.u64 q13, q9,  #5
vshl.u64 q9,  q9,  #10
vshl.u64 q14, q10, #5
vshl.u64 q10, q10, #10
vshl.u64 q15, q11, #5
vshl.u64 q11, q11, #10
vsra.u64 q12, q12, #1
vsra.u64 q8,  q8,  #10
vsra.u64 q13, q13, #1
vsra.u64 q9,  q9,  #10
vsra.u64 q14, q14, #1
vsra.u64 q10, q10, #10
vsra.u64 q15, q15, #1
vsra.u64 q11, q11, #10
vsub.u64 q8,  q8,  q12
vsub.u64 q9,  q9,  q13
vsub.u64 q10, q10, q14
vsub.u64 q11, q11, q15
```

(b) ASM Code 2 (Adjusted)

Fig. 4. Two pieces of code for constructing vector multiplication with 977.

5.2 Performance Comparison of Prime Field Multiplication

We perform the experiments on the Exynos 4412 development board equipped with the Cortex-A9 processor (1.4GHz), and clock cycles are measured by reading counter registers from Performance Monitoring Unit (PMU) inside CP15 co-processor of ARM. We select three elliptic curves over prime fields with different categories and security levels, `NIST P192`, `Secp256k1`, and `Numsp256d1`, to deploy the experiments. We implement the SMCOS, MR [24] and CICOS [27] vector modular multiplication methods in ARM assembly language, and integrate them into several ECC algorithm libraries for comparison. In order to control the variables, specifically, for `NIST P192` curve, we choose `OpenSSL 1.1.1k` [21] to perform the replacements and evaluations of the above three vector methods on corresponding prime field. For `Secp256k1` curve, `libsecp256k1` [23], the fastest official algorithm library used for Bitcoin protocol implementations, is selected for method replacements. It is worth mentioning that its modular multiplication implementation is optimized by manual assembly before. As for `Numsp256d1` curve, we choose the ECC algorithm library `MSR ECCLib 2.0` [18] provided by Microsoft Research.

For our SMCOS method, MR method, CICOS method, and several ECC libraries, Table 3 summarizes the number of clock cycles required to perform one modular multiplication operation on the three curves. This result impressively demonstrates the efficiency of SMCOS for modular multiplication in ECC, and indirectly supports the performance analysis of Section 4.2, that is, SMCOS

Table 3. Comparison of clock cycles for modular multiplication*

Elliptic Curve	Field	Implementation	Mod-Mul
NIST P192	$\mathbb{F}_{2^{192}-2^{64}-1}$	**Our SMCOS**	205
		MR[24]	301
		CICOS[27]	387
		OpenSSL[21]	1,079
Secp256k1	$\mathbb{F}_{2^{256}-2^{32}-977}$	**Our SMCOS**	310
		CICOS[27]	574
		libsecp256k1[23]	434
		OpenSSL[21]	2,051
Numsp256d1	$\mathbb{F}_{2^{256}-189}$	**Our SMCOS**	306
		CICOS[27]	574
		MSR ECClib[18]	1,050

*Entries are clock cycles measured on a ARM Cortex-A9 processor.

uses fewer instructions in the multiplication and reduction stages, and has higher performance.

The detailed results are as follows. For `NIST P192` curve, our SMCOS method only needs 205 clock cycles to complete a modular multiplication operation, which is about 32% faster than MR, about 47% faster than CICOS, and more than five times as fast as the special NIST modular multiplication in `OpenSSL`. For `Secp256k1` curve, the clock cycles of SMCOS is only 310, which is almost equal to the time to conduct a 256-bit multiplication in [27]. This result is roughly 46% faster than CICOS, 29% faster than the hand-optimized modular multiplication of `libsecp256k1`, and 85% faster than the Montgomery method used by `OpenSSL`. For `Numsp256d1` curve, SMCOS also has an overwhelming advantage, about 47% faster than CICOS and about 71% faster than `MSR ECClib`. Moreover, for these ECC algorithm libraries, except for the modular multiplication of `libsecp256k1`, other libraries are implemented in C language on ARM platforms. This is why the performance of these libraries is much lower than that of several vector methods such as SMCOS.

5.3 Performance Comparison of Elliptic Curve Arithmetic

Point addition and point doubling based on underlying prime field arithmetic are the core operations of various ECC protocols. Table 4 shows the clock cycles of point addition and point doubling for each implementation. Profit from the better optimization of the prime field multiplication, the point addition and point doubling using the SMCOS method also gain better experimental results than other methods. In `OpenSSL`'s `NIST P192`, our SMCOS method requires 6340 and 5755 clock cycles to perform point addition and point doubling, which are roughly 18% and 23% faster than MR, roughly 27% and 32% faster than CICOS. As for `Secp256k1`, SMCOS makes point addition and point doubling reach record-setting execution times on Cortex-A9 processors, they only consume 5853 and 2736 clock cycles, which are about 18% and 11% faster than `libsecp256k1`,

also about 35% and 40% faster than CICOS. For `Numsp256d1` curve, the point addition and point doubling on SMCOS require 7657 and 3297 clock cycles, which are about 36% faster than CICOS, and about three times as fast as `MSR ECClib`.

By referring to Table 4, compared to `libsecp256k1` and `MSR ECClib` for 256-bit ECC, the performance of 192-bit `NIST P192` in native `OpenSSL` is lower. So even if we use several vector methods to replace its prime field multiplication, point addition and point doubling do not gain good benchmark results. This is also the reason why we choose efficient dedicated libraries for `Secp256k1` and `Numsp256d1` curves. But even so, deploying our SMCOS method to `OpenSSL` still has a greater performance enhancement than other designs.

Table 4. Comparison of clock cycles for point addition and point doubling*

Elliptic Curve	Implementation	Point Addition	Point Doubling
NIST P192	**Our SMCOS**	6,340	5,755
	MR[24]	7,692	7,409
	CICOS[27]	8,727	8,419
	OpenSSL[21]	17,003	14,860
Secp256k1	**Our SMCOS**	5,853	2,736
	CICOS[27]	9,017	4,563
	libsecp256k1[23]	7,132	3,064
	OpenSSL[21]	25,840	24,751
Numsp256d1	**Our SMCOS**	7,657	3,297
	CICOS[27]	11,893	5,164
	MSR ECClib[18]	19,424	9,220

*Entries are clock cycles measured on a ARM Cortex-A9 processor.

5.4 Performance Results of ECDH and ECDSA

The ultimate goal of our SMCOS method is to reduce the computational complexity of ECC protocols such as ECDSA and ECDH, and improve their performance, so that they can be used more extensively on general-purpose computing devices, especially on embedded devices. As far as the performance of ECDSA and ECDH, to evaluate the impact of our implementation techniques, we compare SMCOS with two fast vector modular multiplication methods, MR and CICOS, and several ECC libraries.

Table 5 shows the benchmark results of ECDH key exchange and ECDSA signature based on several implementations. For ECDH key exchange, the SMCOS method is roughly 27% $\sim$ 34% faster than CICOS (all 3 curves), 20% faster than MR (`NIST P192`), 63% faster than `OpenSSL` (`NIST P192`), 14% faster than `libsecp256k1` (`Secp256k1`), and 63% faster than `MSR ECClib` (`Numsp256d1`). Moreover, ECDSA signature using SMCOS is about 17% $\sim$ 26% faster than CICOS, 17% faster than MR, 58% faster than `OpenSSL`, 10% faster than `libsecp256k1`, and 25% faster than `MSR ECClib`. In summary, ECDSA signature and ECDH

Table 5. Comparison of clock cycles for ECDH and ECDSA*

Elliptic Curve	Implementation	ECDH Key Exchange	ECDSA Signature
NIST P192	**Our SMCOS**	2,282	2,852
	MR[24]	2,855	3,423
	CICOS[27]	3,117	3,672
	OpenSSL[21]	6,206	6,781
Secp256k1	**Our SMCOS**	950	1,291
	CICOS[27]	1,429	1,737
	libsecp256k1[23]	1,103	1,438
	OpenSSL[21]	12,285	13,063
Numsp256d1	**Our SMCOS**	1,401	1,996
	CICOS[27]	2,085	2,418
	MSR ECClib[18]	3,816	2,653

*Entries are 10^3 clock cycles measured on a ARM Cortex-A9 processor.

key exchange based on SMCOS obtain better performance on ARM Cortex-A9 platforms than other methods. There are two main reasons responsible for the results: 1) performing an ECDSA signature or ECDH key exchange often requires thousands or even tens of thousands of modular multiplication operations; 2) the SMCOS multi-precision modular multiplication has better performance than other methods for these ECC implementations.

6 Conclusions

In this paper, we introduce an optimization technique to improve the performance of multi-precision modular multiplication on ARM NEON platforms. More specifically, we propose a design and implementation of prime field multiplication for specific modulus, called SMCOS, to make full use of the computing power of SIMD co-processors for ECC. On the ARM Cortex-A9 platform, our SMCOS method performs modular multiplication of `NIST P192`, `Secp256k1`, and `Numsp256d1` within only 205, 310 and 306 clock cycles, which are roughly 32% faster than MR method of Pabbuleti et al. and about 47% faster than CICOS method of Seo et al.

The SMCOS modular multiplication can be applied to other ECC algorithms as primitives. At the same time, one of the most obvious future work is to apply the proposed modular multiplication routines to Intel-AVX processors.

References

1. Azarderakhsh, R., Liu, Z., Seo, H., Kim, H.: Neon pqcryto: Fast and parallel ring-lwe encryption on arm neon architecture. IACR Cryptol. ePrint Arch. **2015**, 1081 (2015)

2. Bernstein, D.J., Schwabe, P.: Neon crypto. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 320–339. Springer (2012)
3. Bos, J.W., Montgomery, P.L., Shumow, D., Zaverucha, G.M.: Montgomery multiplication using vector instructions. In: International Conference on Selected Areas in Cryptography. pp. 471–489. Springer (2013)
4. Câmara, D., Gouvêa, C.P., López, J., Dahab, R.: Fast software polynomial multiplication on arm processors using the neon engine. In: International Conference on Availability, Reliability, and Security. pp. 137–154. Springer (2013)
5. Cheng, H., Großschädl, J., Tian, J., Rønne, P.B., Ryan, P.Y.: High-throughput elliptic curve cryptography using avx2 vector instructions. In: Selected Areas in Cryptography (2020)
6. Cortex, A.: A9 neon media processing engine technical reference manual revision: r4p1 (2012)
7. Faz-Hernández, A., López, J.: Fast implementation of curve25519 using avx2. In: International Conference on Cryptology and Information Security in Latin America. pp. 329–345. Springer (2015)
8. Faz-Hernández, A., Lopez, J., Dahab, R.: High-performance implementation of elliptic curve cryptography using vector instructions. ACM Transactions on Mathematical Software (TOMS) **45**(3), 1–35 (2019)
9. Grewal, G., Azarderakhsh, R., Longa, P., Hu, S., Jao, D.: Efficient implementation of bilinear pairings on arm processors. In: International Conference on Selected Areas in Cryptography. pp. 149–165. Springer (2012)
10. Gueron, S., Krasnov, V.: Software implementation of modular exponentiation, using advanced vector instructions architectures. In: International Workshop on the Arithmetic of Finite Fields. pp. 119–135. Springer (2012)
11. Hisil, H., Egrice, B., Yassi, M.: Fast 4 way vectorized ladder for the complete set of montgomery curves. IACR Cryptol. ePrint Arch. **2020**, 388 (2020)
12. Holdings, A.: Arm architecture reference manual, armv7-a and armv7-r edition. Arm Holdings (2014)
13. Huang, J., Liu, Z., Hu, Z., Großschädl, J.: Parallel implementation of sm2 elliptic curve cryptography on intel processors with avx2. In: Australasian Conference on Information Security and Privacy. pp. 204–224. Springer (2020)
14. Intel Corporation: Using streaming simd extensions (sse2) to perform big multiplications, application note AP-941, available for download at http://software.intel.com/sites/default/files/14/4f/24960, July 2000
15. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Annual International Cryptology Conference. pp. 104–113. Springer (1996)
16. Longa, P.: Four$\mathbb{Q}$NEON: Faster elliptic curve scalar multiplications on arm processors. In: International Conference on Selected Areas in Cryptography. pp. 501–519. Springer (2016)
17. Márquez, R.C., Sarmiento, A.J.C., Sánchez-Solano, S.: Speeding up elliptic curve arithmetic on arm processors using neon instructions. Revista Ingeniería Electrónica, Automática y Comunicaciones ISSN: 1815-5928 **41**(3), 1–20 (2020)
18. Microsoft Research: MSR Elliptic Curve Cryptography library (MSR ECClib), 2014. Available at: http://research.microsoft.com/en-us/projects/nums.
19. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)
20. Oder, T., Pöppelmann, T., Güneysu, T.: Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2014)

21. OpenSSL. The open source toolkit for ssl, available for download at https://www.openssl.org.
22. Orisaka, G., Aranha, D.F., López, J.: Finite field arithmetic using avx-512 for isogeny-based cryptography. In: Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. pp. 49–56. SBC (2018)
23. P Wuille et al.: libsecp256k1: Optimized c library for ec operations on curve secp256k1. (2015)
24. Pabbuleti, K.C., Mane, D.H., Desai, A., Albert, C., Schaumont, P.: Simd acceleration of modular arithmetic on contemporary embedded platforms. In: 2013 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–6. IEEE (2013)
25. Page, D., Smart, N.P.: Parallel cryptographic arithmetic using a redundant montgomery representation. IEEE Transactions on Computers **53**(11), 1474–1482 (2004)
26. Sánchez, A.H., Rodríguez-Henríquez, F.: Neon implementation of an attribute-based encryption scheme. In: International Conference on Applied Cryptography and Network Security. pp. 322–338. Springer (2013)
27. Seo, H., Liu, Z., Großschädl, J., Choi, J., Kim, H.: Montgomery modular multiplication on arm-neon revisited. In: International Conference on Information Security and Cryptology. pp. 328–342. Springer (2014)
28. Seo, H., Liu, Z., Großschädl, J., Kim, H.: Efficient arithmetic on arm-neon and its application for high-speed rsa implementation. Security and Communication Networks **9**(18), 5401–5411 (2016)
29. Walter, C.D., Thompson, S.: Distinguishing exponent digits by observing modular subtractions. In: Cryptographers' Track at the RSA Conference. pp. 192–207. Springer (2001)
30. Wang, J., Vadnala, P.K., Großschädl, J., Xu, Q.: Higher-order masking in practice: A vector implementation of masked aes for arm neon. In: Cryptographers' Track at the RSA Conference. pp. 181–198. Springer (2015)
31. Zhao, Y., Pan, W., Lin, J., Liu, P., Xue, C., Zheng, F.: Phirsa: exploiting the computing power of vector instructions on intel xeon phi for rsa. In: International Conference on Selected Areas in Cryptography. pp. 482–500. Springer (2016)