Accelerating SM2 Digital Signature Algorithm using Modern Processor Features [⋆]

Long Mai, Yuan Yan, Songlin Jia, Shuran Wang, Jianqiang Wang, Juanru Li[1], Siqi Ma[2], and Dawu Gu[1]

[1] Shanghai Jiao Tong University, China
[2] Data 61, CSIRO, Australia
{root_mx, turing, jsl_713, wshuran}@sjtu.edu.cn, wjq.sec@gmail.com, jarod@sjtu.edu.cn, siqimslivia@gmail.com, dwgu@sjtu.edu.cn

Abstract. The public key cryptographic algorithm SM2 is now widely used in electronic authentication systems, key management systems, and e-commercial applications systems. As an asymmetric cryptographic algorithm is based on elliptic curves cryptographic (ECC), the SM2 algorithm involves many complex calculations and is expected to be sufficiently optimized. However, we found existing SM2 implementations are less efficient due to the lack of proper optimization. In this paper, we propose `Yog-SM2`, an optimized implementation of SM2 digital signature algorithm, that uses features of modern desktop processors such as extended arithmetic instructions and the large cache. `Yog-SM2` utilizes new features provided by modern processors to re-implement functions of big number arithmetic, prime field modular, elliptic curve point calculation, and random number generation. The use of these new hardware features significantly improves the performance of both SM2 signing and verifying. Our experiments demonstrated that the execution speed of `Yog-SM2` exceeds four mainstream SM2 implementations in state-of-the-art cryptographic libraries such as OpenSSL and Intel ippcp. In addition, `Yog-SM2` also achieves a better performance (97,475 sign/s and 18,870 verify/s) against the OpenSSL's optimized implementation of ECDSA-256 (46,753 sign/s and 16,032 verify/s, OpenSSL-1.1.1b x64) on a mainstream desktop processor (Intel i7 6700, 3.4GHz). It indicates that SM2 digital signature is promising in a widespread application scenarios.

Keywords: SM2 digital signature algorithm · Instruction set extensions · Elliptic curve cryptography

[⋆] This work was partially supported by the Key Program of National Natural Science Foundation of China (Grant No.U1636217), the General Program of National Natural Science Foundation of China (Grant No.61872237), the National Key Research and Development Program of China (Grant No.2016QY071401), and the Major Project of Ministry of Industry and Information Technology of China (Grant No.2018-36). We especially thank Ant Financial Services Group for the support of this research within the *SJTU-AntFinancial Security Research Centre*.

1 Introduction

Issued by the State Cryptography Administration of China on December 17th, 2010, SM2 public key cryptographic algorithm is an asymmetric cryptography algorithm based on elliptic curves cryptography (ECC) and can be used to implement digital signature algorithm (DSA), key exchange protocol, and public key encryption. Later, SM2 digital signature algorithm was officially defined as an international standard in ISO/IEC14888-3/AMD1 on November 3rd, 2017. In reality, SM2 has been widely adopted in various application scenarios especially for financial industries (e.g., the bank transaction system [7]), industrial systems (e.g., PetroChina [2]), blockchain [8], and data protection (e.g., video conference program [13]).

Theoretically, elliptic curve based digital signature algorithms not only achieve a better security but also requires a smaller storage compared to the RSA digital signature algorithm. Thus, existing software products are recommended to update their crypto components in using such digital signature algorithms such as SM2DSA (SM2 Digital Signature Algorithm) and ECDSA (Elliptic Curve Digital Signature Algorithm). However, this implementation scheme significantly affects the actual execution speed of digital signature algorithms in real-world crypto libraries. For SM2DSA, it is generally more complex than the state-of-the-art ECDSA due to its structure and chosen parameters. In response, a series of researches [30, 36–38] focus on improving the performance of SM2DSA. Previous researches [32, 39] mainly study how to optimize SM2 at the hardware level. From the perspective of software optimization, Gueron *et al.* [29] aimed at optimizing two crucial operations–point-addition (*PA* for short) and point-doubling (*PD* for short) by implementing them in different coordinates. A more comprehensive work introduced by Brown *et al.* [28] is to optimize *PD* operation in *Jacobian* coordinates, *PA* in mixed *Affine-Jacobian* coordinates, fixed-point scalar multiplication by *comb method with two tables*, and free-point scalar multiplication by *window NAF* (non-adjancent form) method.

Unfortunately, we observed that seldom research considers how to exploit features of modern processors (e.g., instruction set extensions of Intel core and AMD Ryzen) to improve the execution speed of SM2DSA, although such features have being utilized by many state-of-the-art cryptographic libraries to achieve a highly-optimized version of ECDSA. For instance, in the latest OpenSSL (i.e., OpenSSL-1.1.1+), the optimized version of ECDSA executes three times faster than the compatible version (i.e., the one without involving new features of processors), and 23 times faster than its SM2DSA implementation (i.e., the one without involving new features of processors). If similar features of modern processors can be utilized by SM2DSA, the performance is expected to be improved significantly.

In response, in this paper we propose `Yog-SM2`, an optimized SM2DSA implementation by utilizing various hardware features of modern desktop processors. In detail, `Yog-SM2` customizes functions of **big number arithmetic**, **modular operations**, **scalar multiplication**, and **random number generator** using extended arithmetic instructions provided by cutting-edge processors. Through applying such hardware based optimizations, Yog-SM2 achieves a considerable execution speed (97,475 sign/s and 18,870 verify/s) against its counterpart ECDSA of OpenSSL-1.1.1b x64 (46,753 sign/s and 16,032 verify/s) on an Intel i7 6700 processor. Moreover, Yog-SM2 outperforms four mainstream implementations of SM2DSA in state-of-the-art open source cryptographic libraries. To the best of our knowledge, Yog-SM2 is the most efficient SM2DSA implementation with nowadays desktop processors.

In summary, this paper achieves the following three contributions:

– We built Yog-SM2, a processor level optimized SM2DSA implementation. It fully utilizes various features of modern processors and thus significantly reduces the execution overhead of both signature and verification.
– Yog-SM2 re-implement many low-level functions such as a novel fixed-point scalar multiplication scheme which only consumes 31 *PA* operations to implement a 256-bit scalar multiplication for a specify base point on target elliptic curve, and a specific random number generator with only 82 instructions executed. This guarantees that Yog-SM2 is highly compact and efficient.
– Yog-SM2 not only outperforms existing SM2DSA implementations but also ECDSA implementations. The design and implementation of Yog-SM2 are expected to help designers of cryptographic algorithms to optimize other ciphers especially public key ciphers.

2 Background

2.1 SM2 Implementation

SM2 is an elliptic curve public key cryptographic algorithm issued by Chinese State Cryptography Administration on December 17th, 2010 [12]. Later, it was officially included in ISO/IEC14888-3/AMD1 on November 3rd, 2017. SM2 can be used for key-exchanging, data encryption and decryption, digital signature and verification [19–23]. In this paper, we only focus on digital signature.

To implement a SM2DSA algorithm, a series of complex calculations with both big numbers and elliptic curves are required. In practice, a typical implementation of SM2 requires the following functions as illustrated in Figure 1. In the following, we introduce key functions in SM2 implementation:

– **Big Number Arithmetic Functions**: Big number arithmetic functions are fundamental for SM2. Those functions perform the basic arithmetic calculation (e.g., add, mod) of big number (e.g., 256-bit). Cryptographic libraries (e.g., OpenSSL [17] and Botan [5]) often implement their own big number arithmetic functions, or refer to some big number library such as the GNU MP (GMP) Bignum Library [9]. They usually support any length of big number for compatibility.
– **Prime Modular Functions**: Prime modular functions implements the operations of *modular multiplication*, *modular square*, *modular inversion*, etc. In addition, modular operations are usually used with arithmetic operations.

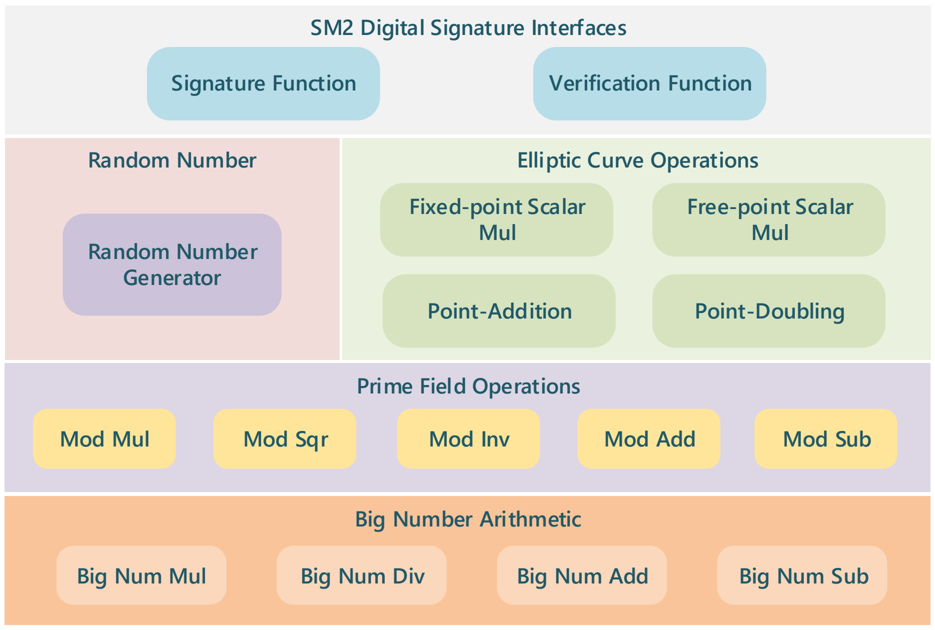


Fig. 1. Overview of SM2DSA algorithm

For example, a modular multiplication contains two operations: a multiplication and a modular reduction. Another feature for prime modular functions is that they will be involved when we convert a point from *Jacobian* coordinates to *Affine* coordinates. Note that most modular functions are time-consuming operations and thus directly influence the performance of both signature and verification functions.

– **Elliptic Curve Functions**: As the basic calculation of Elliptic Curve points, the *PA* and *PD* operations are basis of the elliptic curve point scalar multiplication. Scalar multiplication is classified to two types: *fixed-point scalar multiplication* and *free-point scalar multiplication*. Fixed-point scalar multiplication is used in both signature function and verification function, while free-point scalar multiplication is only used in verification function.
– **Random Number Functions**: The generation of an SM2 digital signature requires a random number to prove its security. To obtain a secure random number, traditional implementations usually need to collect information about the current environment to first generate a seed with high entropy, and then use a pseudo random number generator (PRNG) to extend the seed to a random number (e.g., 256-bit).

In addition to those functions, most SM2 implementations would provide high-level *sign* and *verify* interfaces to help generate digital signatures as well as verify them.

2.2 Features of Modern Processors

Modern processors (e.g., Intel CPUs with Skylake or CoffeeLake micro-architecture) introduces a plenty of new instruction set extensions and hardware features to boost the executions of different programs. First, most modern processors obtain multiple 64/128/256-bit registers and large caches (e.g., 8M L3 cache). These features allow software to load more data into cache and registers to perform complex computation. Especially for vector computation (e.g., multimedia processing technology) and cryptographic algorithm, and those new features can efficiently accelerate their processing procedure and speed the performance of application. Second, each generation of mainstream processors often introduce new instruction set extensions. Except the basic x86/64 instruction set, the latest generation of Intel Core processor (i.e., codename *Coffee Lake*) contains 30 instruction extensions (*MOVBE*, *MMX*, *SSE*, *SSE2*, *SSE3*, *SSSE3*, *SSE4.1*, *SSE4.2*, *POPCNT*, *AVX*, *AVX2*, *AES*, *PCLMUL*, *FSGSBASE*, *RDRND*, *FMA3*, *F16C*, *BMI*, *BMI2*, *VT-x*, *VT-d*, *TXT*, *TSX*, *RDSEED*, *ADX*, *PREFETCHW*, *CLFLUSHOPT*, *XSAVE*, *SGX*, *MPX*). The instruction set extensions cover a diverse range of application domains and programming usages.

3 Yog-SM2

In this section we present `Yog-SM2`, a highly-optimized implementation of the SM2DSA algorithm. `Yog-SM2` fully utilizes several features of modern processors such as Intel Core and AMD Ryzen, and achieves a considerable performance increase in comparison with its counterpart (i.e., the optimized ECDSA in OpenSSL). In detail, `Yog-SM2` leverages both new instruction extensions of modern processors and hardware characteristics (e.g., larger cache) to optimize functions of **big number arithmetic**, **modular operations**, **scalar multiplication**, and **random number generator**. In addition, `Yog-SM2` adopts a **redundant instruction removal** policy to implement instruction-level efficient operations. In the following, we elaborate how `Yog-SM2` implements each optimization.

3.1 Optimization Strategies

Extended Arithmetic Instructions Since the calculations of elliptic curve involves a large number of arithmetic operations, `Yog-SM2` utilizes extended arithmetic instructions to boost the execution (see Section 3.2 and 3.3). In detail, `Yog-SM2` utilizes three instructions including *mulx*, *adcx* and *adox*. Table 1 gives a detailed descriptions about the mentioned instructions. These instructions are alternative versions of existing x86 instruction (*mul*, and *adc*) and fulfil the operations of multiple and addition, respectively. However, these three instructions are designed to support two separate carry chains and thus are used to speed up large integer arithmetic [34]. For instance, the *mulx* instruction does not affect any flag when executing, and the operating results can be saved in any common registers, which is more convenient than the original *mul* instruction. Moreover, this instruction does not overwrite the source operands.

Large Capacity On-chip Storage Nowadays processors often carry larger cache (e.g., 16MB L3 cache) and registers with 64 to 512 bits. These features are leveraged to optimize our `Yog-SM2`. First, `Yog-SM2` adopt a large look-up table (i.e., 512KB) to accelerate the computation of fixed-point scalar multiplication (see Section 3.4). The look-up table contains 8,192 elliptic curve points in *Affine* coordinates. Traditionally, this occurs a frequent memory access and may introduce performance penalty. However, on modern processors the use of this table benefits from the large cache. Second, since most modern processors support 64-bit registers, the arithmetic operations in `Yog-SM2` are fully optimized using 64-bit instead of 32-bit registers. In this way, the calculation is sufficiently boosted.

Table 1. Extended instructions used by `Yog-SM2` to optimize arithmetic operations

Instruction	Instruction set	Description
mulx r64a, r64b, r/m64	BMI2	Unsigned multiply of r/m64 with RDX without affecting arithmetic flags.
adcx r64, r/m64	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.
adox r64, r/m64	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

3.2 Big Number Arithmetic Optimization

To optimize the big number arithmetic, we fully utilized 64-bit registers and the extended arithmetic instructions in modern processors.

Big Number Multiplication When preforming big number multiplication $c = a * b$ (a, b and c are 256 bits number), `Yog-SM2` separates the multiplication procedure into several rounds with each round calculate $a[i]*b[j]$ (i and j between 0 and 3, and each $a[i]$ or $b[j]$ is 64 bits and store in a single 64-bit register). In each round, `Yog-SM2` first performs a multiplication operation with *mulx* instruction, followed by a serial of addition operations with *adcx* or *adox* instruction. Those serial addition operations are to add the product that the *mulx* instruction produces to the final result c. By using new instructions, only the *CF* flag or *OF* flag would be affected, and thus the overhead is much lower.

Big Number Modular When performing modular operation, `Yog-SM2` does not use the traditional "conditional subtraction" method (e.g, calculating $c = a \ mod \ b$ to judge the condition $a > b$, if satisfied, performs a subtraction). Instead, it uses a non-branch and sequential execution method to execute the modular subtraction. The idea for non-branch and sequential execution benefits from two instructions: *cmovz* and *cmovnz* [6]. These two instructions are the variants of *mov* instruction which performs different actions (e.g., move data or not) according to the zero flag `ZF` of `EFLAGS` register. By utilizing these instructions, we avoid the penalty of execution prediction failure and thus make better use of the CPU resources.

Another optimization for modular operation is to merge it with other operations. Aiming at higher performance, merging modular operation with other operations could reduce unnecessary instructions as well as extra memory accesses. For example, `Yog-SM2` implements a big number modular addition operation *modAdd* instead of two separated functions (a modular operation and a addition operation). When it is frequently invoked, the cost of function call and return is reduced from twice to once.

3.3 Modular Operation Optimization

The implementation of `Yog-SM2` adopts optimized modular multiplication and modular inversion, which benefit from the extended arithmetic instructions. Moreover, we improve the Montgomery modular multiplication by integrating multiplication operation into modular operation to reduce memory access operations and increase the efficiency of register usage. At the same time, we inline all sub functions of modular inversion to accelerate its procedure.

Modular Multiplication We optimized the traditional Word-by-Word Montgomery Friendly Multiplication (*WW-MF*) algorithm used in modular multiplication with the *mulx*, *adcx*, and *adox* instructions. For two 256-bit numbers a and b (both can be saved by four 64-bit registers), the calculation of multiplication can be divided into four rounds:

1. Calculate $a * b[0]$, ($b[0]$ is the less signification 64 bits of b)
2. Calculate $a * b[1]$,
3. Calculate $a * b[2]$,
4. Calculate $a * b[3]$, ($b[3]$ is the most signification 64 bits of b)

Here we utilize extended arithmetic instructions to fulfil the multiplication and addition operations. Moreover, we notice that the original *WW-MF* algorithm is first to calculate the multiplication of a and b, storing the result into a temporary 512-bit variable T (which costs eight registers to store it), and then to reduce T from 512 bits to 256 bits by four rounds. In order to optimize the use of registers, we customized this algorithm to integrate four rounds of multiplication operation with four rounds of reduction operation. In other words, we perform one round of reduction operation after one round of multiplication operation. By this way, only **six** instead of **eight** registers are needed to save the intermediate results. Moreover, since the modular P used by SM2 is a Montgomery friendly modular (satisfying $-P^{-1} mod\ 2^s = 1$, s is the word size of current machine, in our environment, $s = 64$), the reduction steps can be further optimized from five steps to four steps.

Modular Inversion To optimize the modular inversion operation of SM2DSA, we re-implement big number (256-bit) *shift-left*, *shift-right*, *addition*, and *subtraction* functions, which are used by the Almost Montgomery Inversion [35] (*AlmMonInv* for short) algorithm, a core algorithm for modular inversion (*AlmMonInv* algorithm is used to support both modular N inversion operation and modular P inversion operation, where N is the order of the base point G in elliptic curve and P is the prime number). We re-implement those functions with *adcx* and *adox* instructions, and inline all sub function in modular inversion to avoid function call and unnecessary memory access operations. In this way, the modular inversion operation is significantly optimized.

3.4 Scalar Multiplication Optimization

The optimization of scalar multiplication is divided into two steps: we first generate a look-up table for fixed-point scalar multiplication and reduce the complexity to only 31 *PA* operations; then we use an adaptive window NAF method to determine the best window for free-point scalar multiplication.

Fixed-Point Scalar Multiplication We proposed a look-up table based fixed-point scalar multiplication that reduces the complexity to exactly 31 *PA* operations for a 256-bit scalar. The signing of SM2 requires a product of G (the base point of the elliptic curve of SM2 algorithm) with a 256-bit random scalar k. By splitting k into 32 bytes ($k = (k_{31}, ..., k_0)$) and pre-computing 256 possible elliptic curve points $P_i = k_i * 2^{8i} * G$, we generate 8,192 pre-computed points as a look-up table. Note that it consumes less storage space to represent the elliptic curve point in *Affine* coordinates than in *Jacobian* coordinates (i.e., 64 bytes for one point). Our table stores each elliptic curve point using the *Affine* coordinates. In total, the size of the look-up table is 512 KB.

When the signing process needs to compute k * G for arbitrary k, it inquires the look-up table according to certain value of k_i and directly obtains the point P_i. Then it only conducts 31 *PA* operations to add these pre-computed points to get the result of k * G. As a result, we speed up the SM2 algorithm substantially.

Free-Point Scalar Multiplication We found that traditional window method `wNAF` to optimize the free-point scalar multiplication fails to set the most optimized parameter (i.e., the window size w) for different processors. In `Yog-SM2`, the best value of w is determined at runtime. `Yog-SM2` will choose different value of w used by `wNAF` and find the best one. In detail, `Yog-SM2` evaluate the following runtime metrics–modular multiplication, modular square, and modular inversion. For instance, on a platform with Intel i7 6700 (3.4GHz) processor, the result shown in Table 2 demonstrates that `Yog-SM2` should set w to 3 to obtain the best performance. In comparison, GmSSL [10] and Intel-ippcp [11] both adopt a `wNAF` with fixed $w = 5$, which are not adaptive to various processors.

Table 2. Performance of `wNAF` method for different window

Window w	**Complexity**			**Running-time (us)**
	M	S	I	
2	1,712	1,282	-	47.85
3	**1,551**	**1,224**	**1**	**45.36**
4	1,477	1,196	3	46.66
5	1,449	1,185	7	56.02

-

M: modular multiplication; S: modular square; I: modular inversion.

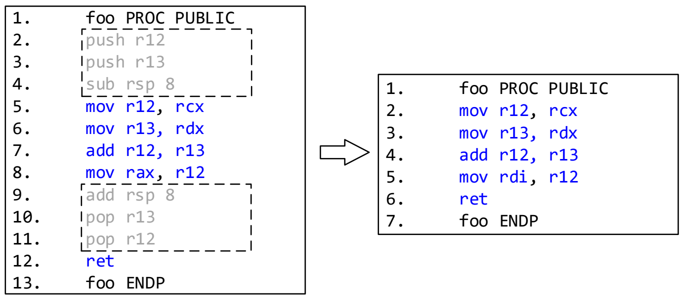
3.5 Random Number Generator

Random number is crucial to SM2 algorithm. During an SM2 signing, it needs a 256-bit random k to help compute the signature. In addition, the random number generator is also used to generate the private key for the digital signature. We observe that generating a random number is usually time-consuming. Traditionally, to generate a (pseudo) random number, a large number of information about the current environment (e.g., memory usage statistics, current process ID, system performance counter, etc.) is collected. Hence, a software pseudo random number generator often executes tens of thousands of instructions to generate a random number.

To optimize the generation of random number, `Yog-SM2` utilizes the Intel *RDRAND* hardware instruction [24] to generate random number for SM2 Signing. *RDRAND* is an instruction to obtain random numbers from an on-chip hardware random number generator. It is part of the Intel 64 and IA-32 instruction set architectures and is available in Intel Ivy Bridge processors and the successors. AMD also added support for the instruction in June 2015. By using this feature, `Yog-SM2` only needs to execute 82 instructions (including the *RDRAND* instruction) to get a 256-bit random number securely.

3.6 Redundant Instruction Removal

We observe that the implementation of *PD* and *PA* functions contain many redundant instructions. We can remove those unnecessary instructions and thus significantly improve the performance. Figure 2 demonstrate a concrete example of redundant instruction removal. In Figure 2(a) the function *foo* follows the convention of parameter passing follows the Windows x64 `Application Binary Interface` (Windows x64 `ABI`) standard [3]. However, its function prologue (top box in dashed line) and function epilogue (bottom box in dashed line) are redundant and can be removed. Usually, function prologue and epilogue are necessary to save and restore the execution context of the caller at the invoking site. In our case, however, if the caller and the callee function (i.e., *(*foo)) DO NOT share registers and stacks. we can remove those unnecessary function prologue and epilogue to reduce memory accesses and thus significantly improve the performance.



```
foo PROC PUBLIC
push r12
push r13
sub rsp 8
mov r12, rcx
mov r13, rdx
add r12, r13
mov rax, r12
add rsp 8
pop r13
pop r12
ret
foo ENDP
```

(a) Original function

```
foo PROC PUBLIC
mov r12, rcx
mov r13, rdx
add r12, r13
mov rdi, r12
ret
foo ENDP
```

(b) Optimized function

Fig. 2. Function prologue and epilogue comparison in assembly form

Figure 2(b) shows the optimized form of the original function. The memory access operations (instructions in the box with dashed line) are removed in the optimized version while the functionality is equivalent to the original version. Note that this redundant instruction removal only works if certain requirements are satisfied: 1) Except for *RSP* and *RIP* registers, all other registers are inherently volatile in callee functions (e.g., function *foo*). The optimized convention gives the callee functions the ability to use any common registers without storing them in function prologue and restoring them in function epilogue. 2) All common registers (except *RSP* and *RIP* registers) should be saved in callers (functions who call the *foo*). Only by this way can we use all registers in callee functions freely. 3) The way to pass parameters is different from the original convention. Fortunately, in our SM2 implementation the *PD* and *PA* functions are suitable for applying such instruction removal. As a result, `Yog-SM2` could benefit from a more compact version of primitive functions.

4 Evaluation

To evaluate `Yog-SM2`, we first analyzed its **computation complexity** and then tested its actual **execution performance**. For computation complexity, we counted numbers of executed modular multiplication (M), modular squaring (S), modular inversion (I), and division (D). For execution performance, we counted instructions executed for signing and verifying, respectively. In addition, we compared the performance of `Yog-SM2` to that of other four libraries including GmSSL, OpenSSL, Botan, and Intel-ippcp.

Our experiments were conducted on a workstation with an Intel core i7 6700 processor (3.4GHz), 16GB DDR4 memory, and 512GB SSD. The operating system is Windows 7 (x64) and the compiler to generate binary code is Visual C++ 2015.

Table 3. Complexity analysis of popular SM2 implementations

Library	**Sign**				**Verify**		
	M	*S*	*I*	*D*	*M*	*S*	*I*
GmSSL-2.5.0	290	109	2	-	2,061	1,401	1
OpenSSL-1.1.1b	3,871	1,802	1	-	2,641	1,569	1
Botan-2.10.0	903	338	2	-	4,105	2,820	1
Intel-ippcp_2019u3	301	109	2	1	2,049	1,397	1
Yog-SM2	**263**	**97**	**2**	-	**1,905**	**1,333**	**1**

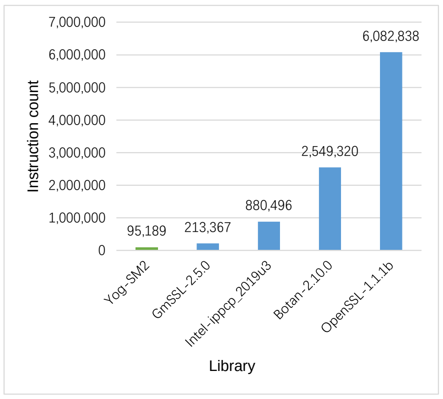
4.1 Complexity Analysis

We first analyzed the computation complexity of SM2 algorithm implemented in Yog-SM2 and other four mainstream cryptographic libraries. The results are shown in Table 3. Apparently, Yog-SM2 is the most efficient implementation for both signature operations and verification operations. Because of the optimized look-up table, fixed-point scalar multiplication reduces the needed calculations. To sign a message, Yog-SM2 only required 263 modular multiplication, 97 modular squaring, and two modular inversion. While performing a signature verification, Yog-SM2 proceeded 1,905 modular multiplication, 1,333 modular squaring, and one modular inversion. We also observed that OpenSSL has the highest complexity for the signature operation. Consider the verification operation, Botan has the highest complexity. By manually inspected their code, we found the root cause of such a high complexity: 1) OpenSSL adopts the Montgomery ladder algorithm [16], a constant time algorithm, to sign messages. This significantly increases the complexity. 2) Botan uses the Binary algorithm for both free-point scalar multiplication and fixed-point scalar multiplication in verification. Although code reuse makes the SM2 implementation of Botan more concise, it raises the computation complexity.

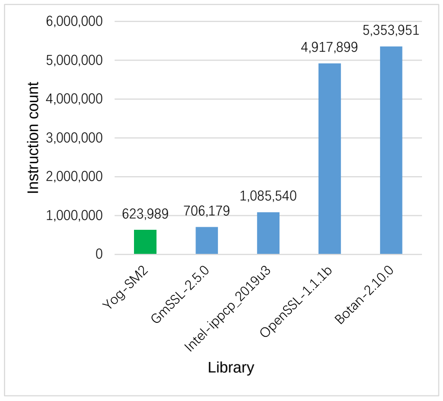
4.2 Execution Performance

We first used the number of executed instructions to evaluate the performance of different SM2 implementations. Results are depicted in Figure 3. Comparing with the other SM2 implementations, Yog-SM2 averagely executed only 95,189 instructions for each signature operation, and 623,989 instructions for a verification operation. The other implementations operated more instructions. Specifically, the instructions executed by the SM2 implementation of OpenSSL is 63.9 times higher for signing and 7.9 times higher for verification. The low efficiency might be caused by its applicable security and compatibility for most platforms.

We also noticed that OpenSSL provides a specialized version of ECDSA that can be used on latest processors. To compare our optimization strategies and that of OpenSSL-ECDSA, we tested Yog-SM2 against OpenSSL's optimized implementation of ECDSA-256 on a mainstream desktop processor (Intel i7 6700, 3.4GHz). Yog-SM2 achieves the speed of 97,475 sign/s and 18,870 verify/s against a 46,753 sign/s and 16,032 verify/s speed of OpenSSL-1.1.1b x64. The result proves



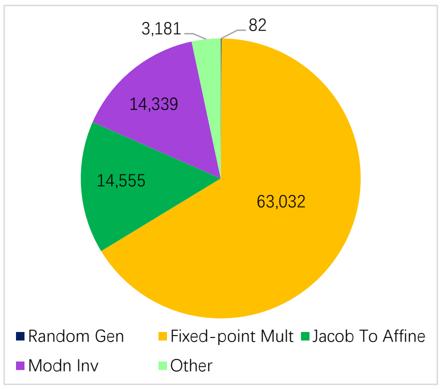
(a) Sign



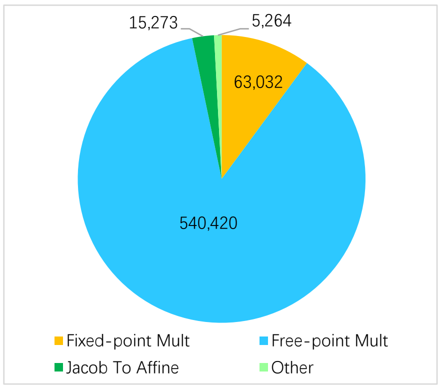
(b) Verify

Fig. 3. Performance comparison of SM2 algorithm for each library

that `Yog-SM2` is also scalable to be extended to a specific platform for performance improvement.



(a) Sign



(b) Verify

Fig. 4. Instruction consumption of each module in `Yog-SM2`

In order to analyze the instruction composition of `Yog-SM2` in detail, we divided `Yog-SM2` into different modules and separately analyzed instruction consumption. Figure 4 describes results of instruction consumption, in which Figure 4(a) and Figure 4(b) show the consumption of signature instruction component and verification component, respectively. For each round of signature, the random number generator of `Yog-SM2` only consumes 82 instructions on average. The fixed-point scalar multiplication operation contains consumption of *PA* operations, which consumes 66.22% of all instructions, The conversion from *Jacobian* coordinates to *Affine* coordinates is also involved whose consumption is almost the same as the modular inversion operation. To execute the modules in verification instruction component, on average, fixed-point scalar multiplication and free-point scalar multiplication executes 10.10% and 86.61% of instructions, respectively. It is worth pointing out that fixed-point scalar multiplication consumes the same for both signature and verification because branches are not created if there is no infinity-point (zero point) for fixed-point scalar multiplication operation.

Comparison of Hardware Improvement To quantitatively measure the effect of using new features in hardware (i.e., modern processors), we compared the performance of `Yog-SM2` with its compatible version–`Yog-SM2/C`. To make `Yog-SM2/C` be suitable for all platforms without utilizing new features in modern processors, we replaced all hardware-dependant instructions with compatible x86 instructions and removed all assembly code. The experiment showed that `Yog-SM2/C` only signs 13,079 times and verifies 1,993 times per second, respectively.

Table 4. Instruction consumption of each core module for `Yog-SM2/C` and `Yog-SM2`

Module Name	`Yog-SM2/C`	`Yog-SM2`	Percentage of Instruction Reduction
Modp Sqr	610	185	69.7%
Modp Mult	694	223	67.9%
Modp Inv	107,170	14,391	86.6%
Random Gen	5,492	82	98.5%
Fixed-point Mult	365,068	62,475	82.9%

The comparison result of `Yog-SM2/C` and `Yog-SM2` is shown in Table 4. The instruction consumption of modular squaring and modular multiplication in `Yog-SM2` reduce 69.7% and 67.9% of instructions while comparing with `Yog-SM2/C`. Modular inversion in `Yog-SM2` consumes 223 instructions, which achieves 86.6% reduction of instruction consumption. For the random number generator, we used *BCryptGenRandom* provided by Microsoft in `Yog-SM2/C` and implemented *rdrand* provided by Intel in `Yog-SM2`. As a result, generating a 256-bit random number costs 5,492 instructions for `Yog-SM/C`, but only 82 instructions for `Yog-SM2`. In total, `Yog-SM2` reduces 98.5% instructions.

We also tested the SM2 implementation of `OpenSSL-1.1.1b` and found the similar result. The SM2 implementation of `OpenSSL-1.1.1b` can only sign 1,988 times and verify 2,326 times per second, respectively. In comparison, the ECDSA-256 (46,753 sign/s and 16,032 verify/s, OpenSSL-1.1.1b x64) achieves a much better speed. Without the accelerating of hardware features, SM2DSA can hardly replace ECDSA in high-performance computation scenarios.

5 Discussion

Since the core operations in `Yog-SM2`, including *PA* operation, *PD* operation and all the called sub functions(e.g., modular multiplication, modular square etc.), were implemented in assembly form, the following optimizations are brought:

– Since the assembly code is not generated relying on source code compilation, a number of unnecessary instructions are eliminated.
– Redundant instructions such as unnecessary *push* and *pop* are removed by the method *redundant instruction removal*.
– As the core operations can fully utilize the extensive register resources by calling new instructions in modern processors in Windows x64 platform, data transfers are mainly carried in registers to speed up the calculation.

Nonetheless, this implementation also causes a scalability limitation. Because the assembly code may vary in different platforms, `Yog-SM2` is hard to be directly applied to another platform. For example, the assembly code for Windows cannot run on Linux directly.

Note that cryptographic libraries (e.g., OpenSSL, Libreswan [14]), that rely on certain hardware features for cryptographic algorithms acceleration [4,15,18], can only be applied to block ciphers and hash function. For instance, in OpenSSL, Intel Advanced Encryption Standard Instructions [1] have been used to accelerate the `AES` algorithm, and Intel SHA Extensions [25] are used for `SHA1` and `SHA-256` algorithms. However, these cryptographic instructions can only be applied to block ciphers and hash functions. Unlike those cryptographic libraries, `Yog-SM2` utilizes a general-purpose hardware feature to optimize public key ciphers.

6 Related Work

SM2 algorithm can be optimized through two aspects, hardware and software. For SM2 hardware optimization, previous works fucused on implementing SM2 algorithm in FPGA and on ASIC chip respectively. Existing software implementation mainly concerned about *PA* and *PD* operations, modular inversion, and modular multiplication operations are commonly used instead.

PA and PD operations. M.Brown *et al.* analyzed the operations *PA* and *PD* in different coordinates. Specifically, they assessed the complexity and running-time overhead of the fixed-point scalar multiplication with difference implementations including binary methods, binary NAF methods, window NAF methods, fixed-based windows methods, and fixed-based comb methods. Nonetheless, new features in modern processors (e.g., large caches) are not utilized. To speed up the operations of *PA* and *PD*, Gueron *et al.* converted the point representation from *Affine* coordinates to *Jacobian* coordinates. The operations *PA* and *PD* are carried to *Jacobian* coordinates, in which the calculations of *PA* and *PD* is faster than those in *Affine* coordinates.

Modular inversion optimization. Kaliski *et al.* [31] proposed a method for modular inversion in Montgomery domain, which helps modular inversion operation avoid trial of division operation. To improve the efficiency of the above mentioned Montgomery modular inversion algorithm, both Savas *et al.* [35] and Sen Xu *et al.* [36] proposed optimized algorithms. Savas *et al.* boosted the second phase of the algorithm. Comparing with the original algorithm, the second phase achieves 6.69 times of increase while giving 160 bits of data and the whole algorithm improves 1.36 times of increase. Besides, Sen Xu *et al.* proposed an efficient constant-time implementation of modular inversion, which relies on the prime field base on the Fermat's little theorem. Their implementation improved 89% of the modular inversion operation for 256 bits prime number.

Modular multiplication operations. Montgomery P.L. [33] proposed an algorithm to calculate modular multiplication without trial division. However, the implementation is not fully optimized. Barrett Paul. [27] proposed Barrett reduction algorithm to reduce a number. The above works are further improved by M.Brown *et al.*. They proposed an algorithm that is suitable for any modular without considering whether a number is a prime. However, this algorithm requires that a product of two numbers are calculated first. The algorithm is then applied to reduce the product, which is not efficient in our case. Adalier *et al.* [26] compared the above mentioned algorithms and concluded that Montgomery modular multiplication algorithm performs the best for 256 bits prime number. Unfortunately, those algorithms do not fully consider characteristics of each modular and new features of modern processors.

7 Conclusion

We present `Yog-SM2`, an optimized implementation of SM2DSA algorithm. `Yog-SM2` utilizes features of modern processors such as extended arithmetic instructions and large cache to fulfil efficient signing and verifying. The evaluation of `Yog-SM2` demonstrated that the performance of SM2 signing and verifying boosts significantly in modern desktop processors such as Intel core i7 processor. Compared with state-of-the-art cryptographic libraries, `Yog-SM2` also achieves better performance with less instructions executed.

References

1. Aes-ni. `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni`
2. Anydef. `http://www.anydef.com/?page_id=18`
3. Application binary interface (abi) for x64. `https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019`
4. Avx instruction acceleration. `https://software.intel.com/en-us/articles/improving-openssl-performance`
5. Botan library. `https://botan.randombit.net/`
6. Cmovcc instruction. `https://www.felixcloutier.com/x86/cmovcc`

7. Dongjin. `http://www.donjin.com/fangan/375.shtml`
8. Fisco bcos. `https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/design/features/guomi.html`
9. Gmp library. `https://gmplib.org/`
10. Gmssl library. `http://gmssl.org/`
11. Intel ippcp library. `https://github.com/intel/ipp-crypto`
12. Issued sm2. `http://www.oscca.gov.cn/sca/xxgk/2010-12/17/content_1002386.shtml`
13. Kedacom. `https://www.kedacom.com/cn/newskd/4800.jhtml`
14. Libreswan library. `https://libreswan.org/`
15. Libreswan library acceleration. `https://libreswan.org/wiki/Cryptographic_Acceleration`
16. Montgomery ladder algorithm. `https://hyperelliptic.org/EFD/g1p/auto-shortw-xz.html#doubling-dbl-2002-it-2`
17. Openssl library. `https://www.openssl.org/`
18. Openssl library acceleration. `https://software.intel.com/en-us/articles/improving-openssl-performance`
19. Public key cryptographic algorithm sm2 based on elliptic curves part 1: General.
20. Public key cryptographic algorithm sm2 based on elliptic curves part 2: Digital signature algorithm.
21. Public key cryptographic algorithm sm2 based on elliptic curves part 3: Key exchange protocol.
22. Public key cryptographic algorithm sm2 based on elliptic curves part 4: Public key encryption.
23. Public key cryptographic algorithm sm2 based on elliptic curves part 5: Parameter definition.
24. Rdrand instruction. `https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`
25. Sha extensions. `https://software.intel.com/en-us/articles/intel-sha-extensions`
26. Adalier, M., et al.: Efficient and secure elliptic curve cryptography implementation of curve p-256. In: Workshop on Elliptic Curve Cryptography Standards. vol. 66 (2015)
27. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 311–323. Springer (1986)
28. Brown, M., Hankerson, D., López, J., Menezes, A.: Software implementation of the nist elliptic curves over prime fields. In: Cryptographers Track at the RSA Conference. pp. 250–265. Springer (2001)
29. Gueron, S., Krasnov, V.: Fast prime field elliptic-curve cryptography with 256-bit primes. Journal of Cryptographic Engineering **5**(2), 141–151 (2015)
30. Hu, X., Zheng, X., Zhang, S., Li, W., Cai, S., Xiong, X.: A high-performance elliptic curve cryptographic processor of sm2 over gf (p). Electronics **8**(4), 431 (2019)
31. Kaliski, B.S.: The montgomery inverse and its applications. IEEE transactions on computers **44**(8), 1064–1065 (1995)
32. Liu, Y., Guo, W., Tan, Y., Wei, J., Sun, D.: An efficient scheme for implementation of sm2 digital signature over gf (p). In: International Conference on E-business Technology and Strategy. pp. 250–258. Springer (2012)
33. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)

34. Ozturk, E., Guilford, J., Gopal, V., Feghali, W.: New instructions supporting large integer arithmetic on intel architecture processors. Intel white paper (2012)
35. Savas, E., Koç, C.K.: The montgomery modular inverse-revisited. IEEE Transactions on Computers **49**(7), 763–766 (2000)
36. Xu, S., Gu, H., Wang, L., Guo, Z., Liu, J., Lu, X., Gu, D.: Efficient and constant time modular inversions over prime fields. In: 2017 13th International Conference on Computational Intelligence and Security (CIS). pp. 524–528. IEEE (2017)
37. Zhang, D., Bai, G.: Ultra high-performance asic implementation of sm2 with power-analysis resistance. In: 2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC). pp. 523–526. IEEE (2015)
38. Zhang, D., Bai, G.: High-performance implementation of sm2 based on fpga. In: 2016 8th IEEE International Conference on Communication Software and Networks (ICCSN). pp. 718–722. IEEE (2016)
39. Zhao, Z., Bai, G.: Ultra high-speed sm2 asic implementation. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. pp. 182–188. IEEE (2014)