PAPER Special Section on Information and Communication System Security

Compact and Efficient Constant-Time GCD and Modular Inversion with Short-Iteration

Yaoan JIN[†a)], *Nonmember and* **Atsuko MIYAJI**[†b)], *Member*

SUMMARY Theoretically secure cryptosystems, digital signatures may not be secure after being implemented on Internet of Things (IoT) devices and PCs because of side-channel attacks (SCA). Because RSA key generation and ECDSA require GCD computations or modular inversions, which are often computed using the binary Euclidean algorithm (BEA) or binary extended Euclidean algorithm (BEEA), the SCA weaknesses of BEA and BEEA become a serious concern. Constant-time GCD (CT-GCD) and constant-time modular inversion (CTMI) algorithms are effective countermeasures in such situations. Modular inversion based on Fermat's little theorem (FLT) can work in constant time, but it is not efficient for general inputs. Two CTMI algorithms, named BOS and BY in this paper, were proposed by Bos, Bernstein and Yang, respectively. Their algorithms are all based on the concept of BEA. However, one iteration of BOS has complicated computations, and BY requires more iterations. A small number of iterations and simple computations during one iteration are good characteristics of a constant-time algorithm. Based on this view, this study proposes new short-iteration CT-GCD and CTMI algorithms over $\mathbb{F}_p$ borrowing a simple concept from BEA. Our algorithms are evaluated from a theoretical perspective. Compared with BOS, BY, and the improved version of BY, our short-iteration algorithms are experimentally demonstrated to be faster.

key words: *constant-time modular inversion (CTMI), constant-time greatest common divisor (CT-GCD), and side channel attacks (SCA).*

1. Introduction

Secure cryptosystems, digital signatures are threatened by side channel attacks (SCA) after being implemented on Internet of Things (IoT) devices and personal computers (PCs). Secret information can be obtained by analyzing various physical parameters including power consumption, implementation time, and electromagnetic emission during the execution of cryptographic algorithms. The endless emergence and improvement of SCA methods make such attacks increasingly dangerous [1], [2].

The binary Euclidean algorithm (BEA) computes the greatest common divisor (GCD). It can be used in RSA key generation to check whether the GCD of the public key e and $\phi(n) = (p-1)(q-1)$, where p and q are randomly generated large primes, is equal to one. The binary extended Euclidean algorithm (BEEA) can be used in RSA key generation and ECDSA to compute a modular inversion. Both algorithms are attractive because they consist of only shift and subtraction operations. However, BEA and BEEA are both threatened by SCA. Specifically, a simple power analysis (SPA), a cache-timing attack (CTA), and a machine learning-based profiling attack (MLPA) were conducted on BEA and BEEA to recover secrets with a high success rate [3]–[7].

A constant-time algorithm can be applied as a countermeasure. Modular inversion based on Fermat's little theorem (FLT) can be computed in constant time. However, it is inefficient for general inputs and can only compute modular inversions over prime numbers. Two constant-time modular inversion (CTMI) algorithms based on the basic concept of BEA were proposed by Bos [8], Bernstein and Yang [9], and are denoted by BOS and BY, respectively. Their algorithms can also compute constant-time GCD (CT-GCD). The number of iterations of CTMI proposed by Bernstein and Yang was improved by Pieter Wuille in [10], which is denoted by hdBY in this paper. However, BY (even hdBY) still requires more iterations than BOS, and the computations during one iteration of BOS are complicated.

In our preliminary work [11], we propose short-iteration constant-time GCD (SICT-GCD) and CTMI (SICT-MI) on $\mathbb{F}_p$ with simple computations in each iteration and a small number of iterations. The core computations (or iteration formula) of SICT-GCD and SICT-MI on $\mathbb{F}_p$ are defined in Definition 1. The iteration formula based on the GCD computations in Lemma 2 consists of only shift and subtraction operations. Theorem 1 shows that the iteration formula converges to $a_n = \mathsf{GCD}(a_0, b_0)$ and $b_n = 0$, where $a_0, b_0 \in \mathbb{Z}$, $a_0 \geq b_0 \geq 0$, and a_0 or b_0 is odd, with limited iterations. Theorem 2 indicates that the necessary number of iterations is $\mathtt{bitlen}(a_0) + \mathtt{bitlen}(b_0)$. Based on Theorems 1 and 2, the number of iterations of the SICT-GCD and SICT-MI algorithms on $\mathbb{F}_p$ is fixed at $2 \cdot \mathtt{bitlen}(a_0)$. We show the SI-GCD and SI-MI algorithms on $\mathbb{F}_p$ in Algorithms 6 and 7, whose computations in each branch are balanced, and then branchless SICT-GCD and SICT-MI on $\mathbb{F}_p$ in Algorithm 8. We theoretically analyze the efficiency of the SICT-GCD and SICT-MI algorithms by comparing the number of iterations and $\mathbb{F}_p$-arithmetic. We also evaluate the efficiency of the SICT-GCD and SICT-MI algorithms on $\mathbb{F}_p$, FLT-CTMI, BOS, BY, and hdBY through experiments.

In this study, we add the analysis of the maximum parallel data flow in an iteration of BOS, BY, hdBY, and our algorithms, which is accessible in a FPGA implementation. It shows that our algorithms are compact and only require short-iteration. We also add the experimental comparisons on primes recommended for use in a post-quantum non-interactive key-exchange protocol (CSIDH) [12]. More-

Manuscript received November 8, 2022.
Manuscript revised March 22, 2023.
Manuscript publicized July 13, 2023.

[†]The authors are with the Graduate School of Engineering, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: jin@cy2sec.comm.eng.osaka-u.ac.jp
b) E-mail: miyaji@comm.eng.osaka-u.ac.jp

DOI: 10.1587/transinf.2022ICP0009

over, considering that modular inversions are used in elliptic curve affine addition formulae, we compare the efficiency of elliptic curve affine addition formulae using different modular inversion methods.

The results show that our SICT-GCD on $\mathbb{F}_p$ saves 7.44%, 13.78%, 16.67%, 18.45%, 24.77%, 27.05% and 28.3% clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit GCD computations, respectively. Our SICT-MI on $\mathbb{F}_p$ saves

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

The remainder of this paper is organized as follows. Related work is introduced in Sect. 2. The SICT-GCD and SICT-MI algorithms for $\mathbb{F}_p$ are described in Sect. 3. In Sect. 4, we evaluate our SICT-GCD and SICT-MI on $\mathbb{F}_p$. Finally, we conclude this paper in Sect. 5.

2. Related Work

2.1 BEA and BEEA

Methods for computing modular inversions can be divided into two categories. The first is based on Fermat's little theorem (FLT), whose basic idea is $a^{-1} = a^{p-2} \bmod p$ for a prime number p ($a \in \mathbb{Z}$ and $\mathsf{GCD}(a, p) = 1$, which is the greatest common divisor of a and p). The other is based on the extended Euclidean algorithm, Algorithm 1, whose basic concept is $\mathsf{GCD}(a, b) = \mathsf{GCD}(b, a \mod b)$ for $a, b \in \mathbb{Z}$. Note that $(b - r)/c$ means a quotient of $b - r$ divided by c in Algorithm 1. The most notable feature of FLT is that it can compute a modular inversion in constant time. By contrast, the extended Euclidean algorithm cannot compute modular inversion in constant time. However, it is more efficient and can be used to compute the greatest common divisor (GCD).

Of the variants of the Euclidean algorithm and extended Euclidean algorithm, the most frequently used variants are the binary Euclidean algorithm and binary extended Euclidean algorithm, which are called BEA and BEEA, respectively. BEA and BEEA, which consist of only shift and subtraction operations, compute GCD and modular inversion, respectively. They are attractive because shift and subtraction operations can more easily be implemented on both the software and hardware. The basic concept of BEA and BEEA is as follows:

$$\mathsf{GCD}(a, b) = \begin{cases} \mathsf{GCD}(|a - b|, \mathtt{min}(a, b)), & a \text{ and } b \text{ are odd} \\ \mathsf{GCD}(a/2, b), & a \text{ is even, } b \text{ is odd} \\ \mathsf{GCD}(a, b/2), & a \text{ is odd, } b \text{ is even} \\ 2 \cdot \mathsf{GCD}(a/2, b/2), & a \text{ and } b \text{ are even.} \end{cases} \quad (1)$$

BEA computes $\mathsf{GCD}(a, b)$ using Eq. (1), as shown in Algorithm 2. For modular inversion, the case that a and b are even does not exist, and the first "while" loop in Algorithm 2 can be removed because $\mathsf{GCD}(a, p) = 1$ when $a^{-1} \bmod p$ exists. BEEA computes the modular inversion of $a^{-1} \bmod p$ ($\mathsf{GCD}(a, p) = 1$), as shown in Algorithm 3. Note that >> indicates the right-shift operation and << indicates the left-shift operation. $A/2 \bmod p$ can be computed as $A >> 1$ when A is even, and $(A + p) >> 1$ when A is odd.

2.2 SCA of BEA and BEEA

A method for recovering the inputs of BEA (or BEEA) was first proposed in [13]. Specifically, the inputs can be recovered by inputting the complete operational flow of an execution of BEA (or BEEA) into the algorithm ([13] Fig. 7). This is because u and v are finally updated to zero and $\mathsf{GCD}(a, b)$, respectively, in Algorithms 2 and 3. With the

Algorithm 1: Extended Euclidean Algorithm

Input: a, p, where $\mathsf{GCD}(a, p) = 1$.
Result: $y_1 = a^{-1} \bmod p$
Initialization:
$b = p$; $c = a$; $x_0 = 1$; $x_1 = 0$; $y_0 = 0$; $y_1 = 1$;
while $c \neq 1$ **do**
  $r = b \bmod c$; $q = (b - r)/c$;
  $b = c$; $c = r$;
  $t = x_1$; $x_1 = x_0 - q \cdot x_1$; $x_0 = t$;
  $t = y_1$; $y_1 = y_0 - q \cdot y_1$; $y_0 = t$;
end
$y_1 = y_1 \bmod p$;
return y_1;

Algorithm 2: BEA

Input: $a, b \in \mathbb{Z}$
Result: $r = \mathsf{GCD}(a, b)$
Initialization:
$u = a$; $v = b$; $r = 1$;
while *u and v are even* **do**
  $u = u >> 1$; $v = v >> 1$; $r = r << 1$;
end
while $u \neq 0$ **do**
  while *u is even* **do**
    $u = u >> 1$;
  end
  while *v is even* **do**
    $v = v >> 1$;
  end
  if $u \geq v$ **then**
    $u = u - v$;
  else
    $v = v - u$;
  end
end
$r = r \cdot v$;
return r

Algorithm 3: BEEA

Input: a, p, where $\mathsf{GCD}(a, p) = 1$
Result: $C = a^{-1} \bmod p$
Initialization:
$u = a$; $v = p$; $A = 1$; $C = 0$;
while $u \neq 0$ **do**
  while u *is even* **do**
    $u = u >> 1$; $A = A/2 \bmod p$;
  end
  while v *is even* **do**
    $v = v >> 1$; $C = C/2 \bmod p$;
  end
  if $u \geq v$ **then**
    $u = u - v$; $A = A - C$;
  else
    $v = v - u$; $C = C - A$;
  end
end
$C = C \bmod p$;
return C

Algorithm 4: FLT-CTMI

Input: a, p, $a \in \mathbb{Z}$, $\mathsf{GCD}(a, p) = 1$ and p is a prime.
Result: $C = a^{-1} \bmod p$
Initialization:
$C = a$; $k = p - 2$ $(\sum_{i=0}^{n-1} k_i 2^i)$;
for $i = n - 2$; $i \geq 0$; $i = i - 1$ **do**
  $C = C^2 \bmod p$;
  if $k_i = 1$ **then**
    $C = C \times a \bmod p$;
  end
end
return C

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

The complete operational flow of an execution of BEA (or BEEA) can be obtained by the SCA. There are some "bad" characteristics in Algorithms 2 and 3, which can be used in SCA to predict the operational flow: 1) There is at most one shift loop of either u or v in each iteration. 2) One subtraction operation must be performed in each iteration. The complete operational flow of the execution of BEEA is obtained by the simple power analysis (SPA) and used to attack the ECDSA in [3]. It is pointed out that the subtraction operational flow can be recovered from the shift operational flow in [3]. Thus, the complete shift operational flow of the execution of BEA (or BEEA) is sufficient to reveal the inputs.

In practical applications, one of the inputs to BEA (or BEEA) is always public. Some characteristics of the inputs to BEA (or BEEA) can be known in advance. All these make SCA to BEA and BEEA easier. Consider RSA key generation in OpenSSL as an example. $\mathsf{GCD}(e, p-1)$ and $\mathsf{GCD}(e, q-1)$, where e is a stable public key and p and q are random large prime numbers with the same bit length, are computed using BEA to check whether p and q are generated properly. The modular inversion of $d = e^{-1} \bmod (p-1)(q-1)$ is computed to obtain the secret key using BEEA. Based on the characteristics of RSA key generation: 1) e and $n = pq$ are public; 2) because $(p-1)(q-1)$ has almost half the same most significant bits (MSBs) as n, the secret $(p-1)(q-1)$ can be recovered by a part of the operational flow at the beginning of the modular inversion computation, $e^{-1} \bmod (p-1)(q-1)$. Referring to n, it is sufficient to recover some least significant bits (LSBs) of $(p-1)(q-1)$ with the operational flow at the beginning and public input e [4], [5]. Benefiting from 1) the general setting of $e = 65537$ is much smaller than $(p-1)(q-1)$, $p-1$ and $q-1$; 2) $4|(p-1)(q-1)$, $2|(p-1)$ and $2|(q-1)$, the partial operational flow at the beginning of BEA (or BEEA) can be predicted in advance. Using all these characteristics, SPA, cache-timing attack (CTA), and machine learning-based profiling attack (MLPA) threaten the security of RSA key generation [4]–[6].

For countermeasures, one can hide the secret inputs of BEA or BEEA using appropriate masking procedures, the security of which was reported in [14]. Another option is to use constant-time GCD (CT-GCD) or constant-time modular inversion (CTMI).

2.3 CT-GCD and CTMI Algorithms

Modular inversion by Fermat's little theorem, which is shown in Algorithm 4, can be computed in constant time because p is the same for each modular inversion, and "if statements" are executed in the same way for any input a. The number of iterations is fixed at $\texttt{bitlen}(p-2)-1$, where $\texttt{bitlen}(p-2)$ is the bit length of $p-2$. The computations in each iteration, which depend on each bit of $p-2$, are identical for any input a. The efficiency of constant-time modular inversion based on FLT, and denoted by FLT-CTMI, depends on the Hamming weight of $p-2$. The larger the Hamming weight of $p-2$, the lower the efficiency of FLT-CTMI. Thus, FLT-CTMI is inefficient for general inputs. Moreover, FLT can compute neither GCD nor modular inversion on a composite number, which is required for RSA key generation, such as $\mathsf{GCD}(e, p-1)$ and $e^{-1} \bmod (p-1)(q-1)$.

Bos proposed a CTMI algorithm by improving Kaliski's algorithm [15] to a constant-time version, which we call BOS [8]. The basic idea is to compute all branches and then select the correct values from them according to the defined signal variables. Because the sum of the bit lengths of the inputs is reduced by one in each iteration, BOS sets the number of iterations to $2 \cdot \texttt{bitlen}(p)$. BOS can compute the modular inversion, Montgomery inversion, and $\mathsf{GCD}(a, b)$, where a or b is odd.

A CTMI was proposed in [16] with inserted dummy computations. The number of iterations is the same as that of BOS. However, inserting dummy computations into CTMI lowers the efficiency and creates potential SCA security issues.

Bernstein and Yang proposed a CTMI algorithm on $\mathbb{F}_p$,

Algorithm 5: BY [9]

Input: $a, p, l = \lfloor (49 \cdot \texttt{bitlen}(p) + 57)/17 \rfloor$,
$pre\_com = 2^{-l} \bmod p$
Result: $q = a^{-1} \bmod p$
Initialization:
$u = a; v = p; q = 0; r = 1; \delta = 1;$
for $i = 0; i < l; i = i + 1$ **do**
  $z = u_{lsb}; s = \texttt{signbit}(-\delta); \delta = 1 + (1 - 2sz)\delta;$
  $u, v = (u + (1 - 2sz)zv) >> 1, v \oplus sz(v \oplus u);$
  $q, r = (q \oplus sz(q \oplus r)) << 1, (1 - 2sz)zq + r;$
end
$q = \texttt{sign}(v) \cdot q;$
$q = q \cdot pre\_com \bmod p;$
return q

Algorithm 5, which we call BY [9]. The iteration formula of their algorithm is shown in Eq. (2), where $\delta \in \mathbb{Z}$ is initialized to one, and the input $b \in \mathbb{Z}$ should be odd.

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

$(\delta, a, b \in \mathbb{Z}, b \text{ is odd.})$

The computations of their algorithm during one iteration are simpler than BOS. Bernstein and Yang analyzed the rate of shrinking of the transition matrix after each iteration and proved that the necessary number of iterations of BY is $\lfloor (49d + 57)/17 \rfloor (d \geq 46)$, where d is the largest bit length of the inputs. The number of iterations is significantly greater than that of BOS. BY can compute the modular inversion, Montgomery inversion, and GCD(a, b), where a or b is odd. In Algorithm 5, `signbit(`a`)` returns zero when $a \geq 0$ or one when $a < 0$, and `sign(`a`)` returns -1 when $a < 0$ or 1 when $a > 0$. The first k iterations of BY are completely determined by the lowest k bits of b and a. This feature supports the divide-and-conquer strategy of BY. Bernstein and Yang also proposed a CTMI algorithm for $\mathbb{F}_{p^n}$, where $\mathbb{F}_{p^n}$ is defined as $\mathbb{F}_p[x]/(f(x))$. Then any element in $\mathbb{F}_{p^n}$ is represented by $g(x) \in \mathbb{F}_p[x]$. The iteration formula is given by Eq. (3), where $f(0) \neq 0$.

$$F(\delta, f, g) = \begin{cases} (1 - \delta, g, (g(0)f - f(0)g)/x) \ \delta > 0, g(0) \neq 0 \\ (1 + \delta, f, (f(0)g - g(0)f)/x) \text{ otherwise.} \end{cases} \quad (3)$$

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

Pieter Wuille kept track of the convex hulls of possible (b, a) after each iteration to find the necessary number of iterations of BY and obtained similar results [10]. He found that the necessary number of iterations of BY for 224-, 256-, and 384-bit computations are 634, 724, and 1086, respectively, whereas Bernstein and Yang showed they are 649, 741, and 1110, respectively. Moreover, Pieter Wuille proposed a variant of BY denoted by hdBY by initializing $\delta = 1/2$ or using the iteration formula in Eq. (4).

$$F(\delta, b, a) = \begin{cases} (2 - \delta, a, \frac{a-b}{2}) \ \delta > 0 \text{ and } a \text{ is odd} \\ (2 + \delta, b, \frac{a+(a \bmod 2)b}{2}) \text{ otherwise.} \end{cases} \quad (4)$$

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

3. Short-Iteration CT-GCD and CTMI

Good characteristics of a constant-time algorithm are short iterations and simple computations during one iteration. We combine the basic concept of BEA and Lemma 1 and propose new short-iteration GCD (SI-GCD) and modular inversion (SI-MI) algorithms, whose computations in each branch are balanced. We then propose CT-GCD and CTMI algorithms, called short-iteration CT-GCD (SICT-GCD) and short-iteration CTMI (SICT-MI), that have short iterations and simple computations.

3.1 Our Iteration Formula

Let us start from a simple lemma of GCD. Lemma 1 implies that the GCD of any two of a, b, and $a - b$, where $a, b \in \mathbb{Z}$ and $a \geq b \geq 0$, is the same. Lemma 1 is used to demonstrate Lemma 2.

Lemma 1. GCD(a, b) = GCD$(b, a - b)$ = GCD$(a, a - b)$, *where* $a, b \in \mathbb{Z}$ *and* $a \geq b \geq 0$.

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

Based on the basic concept of BEA in Eq. (1) and Lemma 1, we show that GCD has the following equivalence relations:

Lemma 2. GCD *has the following equality.*

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

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

Proof.
- Assume that a and b are odd. GCD(a, b) = GCD$(b, a - b)$ by Lemma 1 and GCD$(b, a - b)$ = GCD$(b, (a - b)/2)$ by the concept of BEA.
- Assume that a is even and b is odd. GCD(a, b) = GCD$(a, a - b)$ by Lemma 1 and GCD$(a, a - b)$ = GCD$(a/2, a - b)$ by the concept of BEA.
- Assume that a is odd and b is even. GCD(a, b) = GCD$(b, a - b)$ by Lemma 1 and GCD$(b, a - b)$ = GCD$(b/2, a - b)$ by the concept of BEA.

□

Lemma 2 gives rise to the iteration formula, which is the core computation in the proposed SICT-GCD and SICT-MI.

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

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

Definition 2. $(a, b) = \texttt{max.min}(a, b)$, *where* $a, b \in \mathbb{Z}$, *is defined as follows:*

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

Lemma 3. *Let* $a_n = d \in \mathbb{Z}^+$ *and* $b_n = 0$, *where* d *is odd, be the inputs of* f *in Definition 1. Then, regardless of the number of* f *iterations, the outputs are the same as the inputs, which are* $a_{n+i} = d$, $b_{n+i} = 0$, $i \geq 1$. *Therefore,* $f(f \cdots f(f(a_n, b_n))) = (a_n, b_n)$.

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

Lemma 4. *The sequence* $\{\mathbb{Z} \ni (a_i + b_i) > 0\}$ *is a monotonically decreasing sequence for every two steps, where* a_0 *and* b_0 *are the initial inputs, and* a_i *and* b_i *(*$\mathbb{Z} \ni i > 0$*) are updated by the iteration formula in Definition 1.*

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

(1) After $\texttt{Branch}_1$, $a_{i+1} + b_{i+1} < a_i + b_i$ holds in the Eq. (5).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{a_i - b_i}{2} + b_i) - (a_i + b_i) \\ &= -\frac{a_i + b_i}{2} < 0 \end{aligned} \tag{5}$$

(2) After $\texttt{Branch}_2$ when $4b_i > a_i \geq b_i$, $a_{i+1} + b_{i+1} < a_i + b_i$ holds in Eq. (6).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{a_i}{2} + a_i - b_i) - (a_i + b_i) \\ &= \frac{a_i - 4b_i}{2} < 0 \end{aligned} \tag{6}$$

(3) After $\texttt{Branch}_2$ when $a_i \geq 4b_i$, $a_{i+2} + b_{i+2} < a_i + b_i$. $a_{i+1} = a_i - b_i, b_{i+1} = a_i/2$ are updated because of $a_i - b_i - a_i/2 = (a_i - 2b_i)/2 > 0$. a_{i+1} is odd because a_i is even and b_i is odd. Then a_{i+2}, b_{i+2} are computed by $\texttt{Branch}_1$ and $a_{i+2} + b_{i+2} = (3a_i - 2b_i)/4$, when $a_i/2$ is odd. a_{i+2}, b_{i+2} are computed by $\texttt{Branch}_3$ and $a_{i+2} + b_{i+2} = (3a_i - 4b_i)/4$, when $a_i/2$ is even. Finally, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Eq. (7).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= -\frac{a_i + 6b_i}{4} < 0 \text{ when } a_i/2 \text{ is odd.} \\ &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= -\frac{a_i + 8b_i}{4} < 0 \text{ when } a_i/2 \text{ is even} \end{aligned} \tag{7}$$

(4) After $\texttt{Branch}_3$, $a_{i+1} + b_{i+1} \leq a_i + b_i$ holds in Eq. (8).

$$\begin{aligned} &(a_{i+1} + b_{i+1}) - (a_i + b_i) \\ &= (\frac{b_i}{2} + a_i - b_i) - (a_i + b_i) \\ &= -\frac{3b_i}{2} \leq 0 \ (=0 \text{ when } b_i = 0) \end{aligned} \tag{8}$$

(5) After $(\texttt{Branch}_1, \texttt{Branch}_2)$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Eq. (9).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= (\frac{a_i - b_i}{4} + \frac{a_i - 3b_i}{2}) - (a_i + b_i) \\ &= -\frac{a_i + 11b_i}{4} < 0 \end{aligned} \tag{9}$$

(6) After $(\texttt{Branch}_3, \texttt{Branch}_2)$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Eq. (10).

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= (\frac{b_i}{4} + \frac{3b_i - 2a_i}{2}) - (a_i + b_i) \\ &= \frac{3b_i - 8a_i}{4} < 0 \end{aligned} \tag{10}$$

(7) After $(\texttt{Branch}_2, \texttt{Branch}_2)$ when $2b_i \geq a_i \geq b_i$, $a_{i+2} + b_{i+2} < a_i + b_i$ holds in Eq. (11). Note that there is no pattern of $(\texttt{Branch}_2, \texttt{Branch}_2)$ when $a_i > 2b_i$.

$$\begin{aligned} &(a_{i+2} + b_{i+2}) - (a_i + b_i) \\ &= (\frac{a_i}{4} + \frac{2b_i - a_i}{2}) - (a_i + b_i) \\ &= -\frac{5a_i}{4} < 0 \end{aligned} \tag{11}$$

In summary, all cases are proven as shown in Table 1. □

Theorem 1. *(Convergence) After sufficient iterations of the iteration formula defined in Definition 1, any valid inputs* a_0 *and* b_0 *converge to* $a_n = d$ *and* $b_n = 0$, *where* $d = \mathsf{GCD}(a_0, b_0)$ *and* d *is odd because* a_0 *and/or* b_0 *are/is odd.*

Proof. By Definition 1 and Lemmas 2, 3, and 4, Theorem 1 can be proven. □

As shown in Theorem 1, after sufficient iterations of the iteration formula, $a_n = \mathsf{GCD}(a_0, b_0)$ and $b_n = 0$ are the outputs. Then, we construct our SICT-GCD and SICT-MI by determining the necessary number of iterations.

Table 1 Proved case

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

3.2 The Number of Iterations

We have already shown our iteration formula in Definition 1, which has simple and identical computations (a shift and a subtraction) in each iteration. According to Theorem 2, the iteration formula converges after $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ iterations. Thus, the number of iterations are 448, 512, and 768 for 224-, 256-, and 384-bit GCD computations and modular inversions, respectively. Our number of iterations is fewer than that of hdBY [10] and the same as that of BOS [8]. The required number of iterations is not intuitive in our iteration formula because the decrease in the total bit length of the inputs a_i and b_i after $\texttt{Branch}_2$ cannot be observed directly.

Lemma 5. *For any valid a_i and b_i, $\texttt{bitlen}(a_i)+\texttt{bitlen}(b_i)$ is reduced by at least one after $\texttt{Branch}_1$, $\texttt{Branch}_2$ ($2b_i \geq a_i \geq b_i$) and $\texttt{Branch}_3$ of the iteration formula in Definition 1.*

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

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

Lemma 6. *Assume that even a_i has x bits, odd b_i has y bits, and $a_i > 2b_i$, $x - y \geq 1$. Then $\texttt{bitlen}(a_i) + \texttt{bitlen}(b_i)$ is reduced by at least $x - y + 1$ bits after $x - y + 1$ iterations.*

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

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

Algorithm 6: SI-GCD

Input: $a, b \in \mathbb{Z}$, $a \geq b \geq 0$, a or b is odd. $l = 2 \cdot \texttt{bitlen}(a)$.
Result: $v = \mathsf{GCD}(a, b)$
Initialization:
$u = b$; $v = a$;
for $i = 0$; $i < l$; $i = i + 1$ **do**
  $t_1 = v - u$;
  if *u is even* **then**
    $u = u >> 1$; $(v, u) = \texttt{max.min}(u, t_1)$;
  else
    if *v is odd* **then**
      $t_1 = t_1 >> 1$; $(v, u) = \texttt{max.min}(t_1, u)$;
    else
      $v = v >> 1$; $(v, u) = \texttt{max.min}(v, t_1)$;
    end
  end
end
return v

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

Theorem 2. *After $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ iterations of the iteration formula defined in Definition 1, the valid inputs (a_0, b_0) converge to $(a_n = d, b_n = 0)$, where $\mathsf{GCD}(a_0, b_0) = d$ and d is odd.*

Proof. Theorem 1 shows that the valid inputs (a_0, b_0) converge to $(a_n = d, b_n = 0)$, where $\mathsf{GCD}(a_0, b_0) = d$ and d is odd with sufficient iterations. By Lemmas 5 and 6, $\texttt{bitlen}(a_0) + \texttt{bitlen}(b_0)$ is reduced by at least one after each iteration on average. Thus, the iteration formula converges after at most $bitlen(a_0) + bitlen(b_0)$ iterations. □

Using the iteration formula in Definition 1 and the required number of iterations, the SI-GCD algorithm is shown in Algorithm 6. It is not difficult to determine the transition matrices for a_i and b_i. The SI-MI algorithm is presented according to the transition matrices in Algorithm 7. The computations for each iteration consist of two shifts and two subtractions in Algorithm 7. Note that all the transition matrices are multiplied by two to eliminate multiplications by $1/2$. Thus, we need to multiply the result by $2^{-l} \mod p$, which can be precomputed, where l is the number of iterations.

Montgomery inversion, $a^{-1} \times R \bmod p$, can be computed using SI-MI, where R is generally $2^{\texttt{bitlen}(p)}$ [17]. By setting $pre\_com = 2^{-\texttt{bitlen}(p)} \bmod p$, the output of Algorithm 7 is $a^{-1} \times 2^{\texttt{bitlen}(p)} \bmod p$. For different choices of R, one can change pre_com and obtain $a^{-1} \times R \bmod p$ using Algorithm 7.

For clarity, the computational flow in one iteration of Algorithm 7 can be described as follows:

(1) Update $t_1 = v - u$ and $t_2 = q - r$.
(2) According to the LSB of u and the LSB of v, select the $\texttt{Branch}_i$.

Algorithm 7: SI-MI

Input: $a, p \in \mathbb{Z}$, $p > a > 0$, $\mathsf{GCD}(a, p) = 1$. $l = 2 \cdot \mathtt{bitlen}(p)$.
$pre\_com = 2^{-l} \bmod p$.
Result: $q = a^{-1} \bmod p$
Initialization:
$u = a$; $v = p$; $q = 0$; $r = 1$;
for $i = 0$; $i < l$; $i = i + 1$ **do**
  $t_1 = v - u$; $t_2 = q - r$;
  if *u is odd, and v is odd* **then**
    $t_1 = t_1 >> 1$; $r = r << 1$;
    if $u > t_1$ **then**
      $q = r$; $r = t_2$; $v = u$; $u = t_1$;
    else
      $q = t_2$; $r = r$; $v = t_1$; $u = u$;
    end
  else if *u is odd, and v is even* **then**
    $v = v >> 1$; $t_2 = t_2 << 1$;
    if $t_1 > v$ **then**
      $r = q$; $q = t_2$; $u = v$, $v = t_1$;
    else
      $q = q$; $r = t_2$; $v = v$; $u = t_1$;
    end
  else
    $u = u >> 1$; $t_2 = t_2 << 1$;
    if $t_1 > u$ **then**
      $r = r$; $q = t_2$; $u = u$; $v = t_1$;
    else
      $q = r$; $r = t_2$; $v = u$; $u = t_1$;
    end
  end
end
$q = q \times pre\_com \bmod p$;
return q

Table 2 Data matrix of u and v.

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

Table 3 Data matrix of q and r.

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

(3) According to the selected $\mathtt{Branch}_i$, right shift the data in the second column of Table 2 and left shift the data in the third column of Table 3.
(4) According to the selected $\mathtt{Branch}_i$, compare the data in the second column of Table 2 and the data in the third column of Table 2. Then update u, v, q, and r by the results of the comparison.

3.3 SICT-GCD and SICT-MI

Because each branch in Algorithms 6 and 7 has the same computations, the conditional statements can be removed, as shown in Algorithm 8. Note that this is not the only way to remove conditional statements. For instance, t_1 and t_2 can also be updated as $t_1 = szu \oplus (\bar{s} + \bar{z})(v - u)$ and $t_2 = [sz(v - u) \oplus s\bar{z}v \oplus \bar{s}zu] >> 1$, using three additional XOR operations on F_p and one less addition on F_p. SICT-GCD can be easily obtained by removing the computations

Algorithm 8: SICT-MI

Input: $a, p \in \mathbb{Z}$, $p > a > 0$, $\mathsf{GCD}(a, p) = 1$. $l = 2 \cdot \mathtt{bitlen}(p)$.
$pre\_com = 2^{-l} \bmod p$.
Result: $q = a^{-1} \bmod p$
Initialization:
$u = a$; $v = p$; $q = 0$; $r = 1$;
$sort_1[2] = \{t_1, t_2\}$; $sort_2[2] = \{t_3, t_4\}$;
for $i = 0$; $i < l$; $i = i + 1$ **do**
  $s = u_{lsb}$; $z = v_{lsb}$;
  $t_1, t_2 = (s \oplus z)v + ((sz << 1) - 1)u, [sv + (2 - (s << 1) - z)u] >> 1$;
  ($sz << 1$ is the same as $2sz$.)
  $t_3, t_4 = [(s \oplus z)q + ((sz << 1) - 1)r] << 1, sq + (2 - (s << 1) - z)r$;
  $s = \mathtt{cmp}(t_2, t_1)$; $z = !s$;
  $v = sort_1[s]$; $u = sort_1[z]$;
  $q = sort_2[s]$; $r = sort_2[z]$;
end
$q = q \times pre\_com \bmod p$;
return q

of q and r from Algorithm 8. Thus, we have omitted its description. Function $\mathtt{cmp}(a, b)$ returns one if $a \geq b$ and returns zero if $a < b$. There are two secrets-independent table look-ups of $sort_1[2] = \{t_1, t_2\}$ and $sort_2[2] = \{t_3, t_4\}$, which are shown in Algorithm 8, according to the comparison results of t_1, t_2 and t_3, t_4. Without a loss of generality, we assume that the result of the comparison is $t_2 < t_1$, which can occur in any branch. The operational flow mentioned in Sect. 2.2 cannot be determined based on this information.

Table 4 shows the comparison between FLT-CTMI, BOS [8], BY [9], hdBY [10] and our algorithms, where S, M, xor, add, sub, and shift represent a square, a multiplication, an xor, an addition, a subtraction, and a shift on $\mathbb{F}_p$, respectively. Table 5 shows the number of iterations for 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit computations of FLT-CTMI, BOS, BY, hdBY and our algorithms. FLT-CTMI can only be used to compute modular inversion over prime numbers and is inefficient for general inputs. By contrast, BOS, BY, hdBY and SICT-GCD can compute $\mathsf{GCD}(a, b)$, where a and/or b are/is odd. BOS, BY, hdBY and SICT-MI can compute the Montgomery inversions and modular inversion over any number if it exists. BOS has the same number of iterations as our algorithms. However, its computations, not only the computations on F_p but also the computations of control signals in one iteration, are more complicated than ours (Algorithm 2 in [8]). The computations of BY and hdBY in one iteration are similar to ours. However, their number of iterations, even that of hdBY, is larger than ours. Actually, SICT-GCD and SICT-MI save 13.35%, 13.22%, 13.22%, 13.24%, 13.19%, 13.19 and 13.18% of the number of iterations for 224-, 256-, 384-, 511-, 1020-, 1790- and 2048-bit computations of hdBY, respectively.

We analyze the maximum parallel data flow in one iteration of BOS, BY, hdBY, SICT-MI, as shown in Tables 6-8, where the computations in each row can be computed in parallel. Along with the assumption that an addition, a subtraction, and a shift on $\mathbb{F}_p$ are more costly and omitting the

Table 4 Comparison of CTMI.

	#iterations	Computations in one iteration
FLT-CTMI	$\texttt{bitlen}(p-2)-1$	S (or $S+M$)
BOS [8]	$2\cdot\texttt{bitlen}(p)$	add + 2sub + 6shift
BY [9]	$\lfloor(49\texttt{bitlen}(p)+57)/17\rfloor$	4xor + 2add + 2shift
hdBY [10]	$\texttt{max}(2\lfloor(2455\log_2(M)+1402)/1736\rfloor,$ $2\lfloor(2455\log_2(M)+1676)/1736\rfloor-1),$ $M\geq 157, 0\leq g\leq f\leq M$	4xor + 2add + 2shift
SICT-MI	$2\cdot\texttt{bitlen}(p)$	4add + 2shift

Table 5 The number of iterations of CTMI for 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit computations.

	224-bit	256-bit	384-bit	511-bit	1020-bit	1790-bit	2048-bit
FLT-CTMI	223	255	383	510	1019	1789	2047
BOS [8]	448	512	768	1022	2040	3580	4096
BY [9]	634	724	1086	1446	2886	5064	5794
hdBY [10]	517	590	885	1178	2350	4124	4718
SICT-MI	448	512	768	1022	2040	3580	4096

Table 6 The maximum parallel data flow in one iteration of BOS (Algorithm 2 in [8]).

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

Table 7 The maximum parallel data flow in one iteration of BY and hdBY [9] (Algorithm 5).

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

other computations, all BOS, BY, hdBY, SICT-MI use an addition and a shift on $\mathbb{F}_p$ in an iteration considering the maximum parallel data flow. However, BOS is not compact, where two subtractions, one addition, and four shifts on $\mathbb{F}_p$ are computed in parallel; BY and hdBY uses more iterations. By contrast, our algorithms are compact and with short-iteration.

Table 8 The maximum parallel data flow in one iteration of SICT-MI (Algorithm 8).

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

4. Experiments Analysis

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

We implement BOS, BY, hdBY, and SICT-GCD to compare the efficiency of GCD computations. In our experiments, to compute the GCD, we generate seven sets, each with 10^5 random odd numbers. The numbers in each set are 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit. We compute the GCD of the numbers in each of the sets and $P224-1$, $P256-1$, $P384-1$, $P512-1$, $P1024-1$, $P1792-1$, $P2048-1$. Here, P224, P256, P384 are national institute of standards and technology (NIST) primes, P512, P1024, and P1792 are primes recommended for use in CSIDH [12], and P2048 is a random prime number of 2048 bits. Subsequently, the average clock cycles are measured by `rdtsc`.

Table 9 Comparison of average clock cycles for GCD computation.

	224 bits	256 bits	384 bits	511 bits	1020 bits	1790 bits	2048 bits
BOS [8]	163085	190734	294382	402489	885569	1717641	2016499
BY [9]	107117	134523	211943	290765	667642	1328249	1591591
hdBY [10]	91997	116523	184972	256455	602059	1217655	1454694
SICT-GCD	85156	100464	154142	209138	452901	888231	1042982

Table 10 Comparison of average clock cycles for modular inversion.

	224 bits	256 bits	384 bits	511 bits	1020 bits	1790 bits	2048 bits
FLT-CTMI	206676	154477	389729	615544	2705522	10754437	14650073
BOS [8]	311512	356102	559549	780402	1744711	3397150	4046376
BY [9]	262478	295822	481934	666601	1527635	3060803	3675057
hdBY [10]	223057	248520	400410	565481	1294404	2570412	3067074
SICT-MI	184230	208553	325659	471916	1075802	2119803	2522420

Table 11 Comparison of average clock cycles for ADD+DBL.

	224 bits	256 bits	384 bits
FLT-CTMI	429307	320123	816605
BOS [8]	654161	746821	1159968
BY [9]	532697	602731	972908
hdBY [10]	455161	507967	810430
SICT-MI	379539	430898	667217

The results are presented in Table 9.

From Table 9, it is clear that our proposed SICT-GCD is more efficient than BOS, BY, and hdBY. SICT-GCD saves 7.44%, 13.78%, 16.67%, 18.45%, 24.77%, 27.05% and 28.3% of the clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit GCD computations, respectively.

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

From Table 10, it is evident that FLT-CTMI is the most efficient for 256-bit modular inversions because of the small Hamming weight of P256 − 2, which is 128. SICT-MI is the most efficient CTMI for 224-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions. Shorter iterations of SICT-MI work and it saves 17.41%, 16.08%, 18.67%, 16.55%, 16.89%, 17.53%, and 17.76% on the clock cycles of hdBY on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively. Simpler computations in one iteration of SICT-MI work and it saves 10.86%, 16.44%, 23.33%, 60.24%, 80.29%, and 82.78% on the clock cycles of FLT-CTMI on 224-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively. It saves 40.86%, 41.43%, 41.8%, 39.53%, 38.34%, 37.60%, and 37.66% on the clock cycles of BOS on 224-, 256-, 384-, 511-, 1020-, 1790-, and 2048-bit modular inversions, respectively.

Modular inversions are used in the elliptic curve cryptography, for instance in the elliptic curve affine addition formulae. We compute 10^5 sets of an affine addition formula and anh affine double formula (ADD + DBL) with FLT-CTMI, BOS, BY, hdBY, and SICT-MI and evaluate the average clock cycles. The elliptic curves used in our experiments are NIST elliptic curves, NIST-224, NIST-256, and NIST-384, with their base points. The results are presented in Table 11. ADD + DBL with SICT-MI outperformed all of the other algorithms except for FLT-CTMI on 256-bit. It saves 16.61%, 15.17%, and 17.67% on the clock cycles of that with hdBY on 224-, 256-, and 384-bit, respectively. It saves 11.59% and 18.29% on the clock cycles of that with FLT-CTMI on 224- and 384-bit, respectively. It saves 41.98%, 42.30%, and 42.48% on the clock cycles of that with BOS on 224-, 256-, and 384-bit, respectively.

5. Conclusion

To establish constant-time algorithms that compute the GCD and modular inversion with short iterations and simple computations during one iteration, we defined the iteration formula in Definition 1. We proved that the iteration formula converges to $a_n = \mathsf{GCD}(a_0, b_0)$ and $b_n = 0$ with limited iterations in Theorem 1 and that the required number of iterations of the iteration formula is $\mathtt{bitlen}(a_0) + \mathtt{bitlen}(b_0)$ in Theorem 2. Based on these, we proposed SICT-GCD and SICT-MI algorithms and compared their number of iterations and computations during one iteration with those of FLT-CTMI, BOS, BY, and hdBY. Finally, we experimentally evaluated the efficiency of the SICT-GCD and SICT-MI algorithms in comparison to the FLT-CTMI, BOS, BY, and hdBY algorithms. The results indicate that the SICT-GCD and SICT-MI algorithms are more efficient than hdBY. The number of iterations of hdBY is approximately 2.3 times the bit length of the larger input, and the number of iterations of SICT-GCD and SICT-MI algorithms is 2 times the bit length of the larger input. Thus, the longer the bit length of the larger input, the more iterations and time saved by our algorithms. We plan to perform evaluations on field-programmable gate array (FPGA) in the future.

Acknowledgements

This work was partially supported by JSPS KAKENHI Grant Number JP21H03443, Innovation Platform for Society 5.0 at MEXT, SECOM Science and Technology Foundation and JST Next Generation Researchers Challenging

Research Program JPMJSP2138.

References

[1] S. Chari, J.R. Rao, and P. Rohatgi, "Template attacks," International Workshop on Cryptographic Hardware and Embedded Systems, vol.2523, pp.13–28, Springer, 2003.
[2] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," 23rd USENIX Security Symposium (USENIX Security 14), pp.719–732, 2014.
[3] A.C. Aldaya, A.J.C. Sarmiento, and S. Sánchez-Solano, "SPA vulnerabilities of the binary extended Euclidean algorithm," Journal of Cryptographic Engineering, vol.7, no.4, pp.273–285, 2017.
[4] A.C. Aldaya, R.C. Márquez, A.J.C. Sarmiento, and S. Sánchez-Solano, "Side-channel analysis of the modular inversion step in the RSA key generation algorithm," International Journal of Circuit Theory and Applications, vol.45, no.2, pp.199–213, 2017.
[5] A.C. Aldaya, C.P. García, L.M.A. Tapia, and B.B. Brumley, "Cache-timing attacks on RSA key generation," Cryptology ePrint Archive, 2018.
[6] S. de la Fe, H.-B. Park, B.-Y. Sim, D.-G. Han, and C. Ferrer, "Profiling attack against RSA key generation based on a Euclidean algorithm," Information, vol.12, no.11, p.462, 2021.
[7] S. Xu, X. Lu, A. Chen, H. Zhang, H. Gu, D. Gu, K. Zhang, Z. Guo, and J. Liu, "To construct high level secure communication system: CTMI is not enough," China Communications, vol.15, no.11, pp.122–137, 2018.
[8] J.W. Bos, "Constant time modular inversion," Journal of Cryptographic Engineering, vol.4, no.4, pp.275–281, 2014.
[9] D.J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp.340–398, 2019.
[10] W. Pieter, M. Gregory, and roconnor blockstream, "Safegcd-bounds," Github, 2021.
[11] Y. Jin and A. Miyaji, "Short-iteration constant-time GCD and modular inversion," International Conference on the 21st Smart Card Research and Advanced Application Conference, vol.13820, pp.82–99, 2023.
[12] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: an efficient post-quantum commutative group action," International Conference on the Theory and Application of Cryptology and Information Security, vol.11274, pp.395–427, Springer, 2018.
[13] A. Onur, G. Shay, and J.P. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures," IMA International Conference on Cryptography and Coding, pp.185–203, Springer, 2007.
[14] A. Duc, S. Faust, and F.-X. Standaert, "Making masking security proofs concrete (or how to evaluate the security of any leaking device), extended version," Journal of Cryptology, vol.32, no.4, pp.1263–1297, 2019.
[15] B.S. Kaliski, "The Montgomery inverse and its applications," IEEE Trans. Comput., vol.44, no.8, pp.1064–1065, 1995.
[16] S. Sarna and R. Czerwinski, "RSA and ECC universal, constant time modular inversion," AIP Conference Proceedings, p.050004, AIP Publishing LLC, 2021.
[17] P.L. Montgomery, "Modular multiplication without trial division," Mathematics of computation, vol.44, no.170, pp.519–521, 1985.

Yaoan Jin He received the bachelor's degree of computer science from Shanghai Jiao Tong University in 2017 and the master's degree of electrical and electronic information engineering from Osaka University in 2019. He will receive the PhD degree of electrical and electronic information engineering from Osaka University in 2023. He received the 85th CSEC Outstanding Research Award in 2019. His research work is around Elliptic Curves. He is also interested in secure protocols, FPGA programming, machine learning and try to find out association study between them.

Atsuko Miyaji received the B. Sc., the M. Sc., and the Dr. Sci. degrees in mathematics from Osaka University,in 1988, 1990, and 1997 respectively. She is an IPSJ fellow. She joined Panasonic Co., LTD from 1990 to 1998 and engaged in research and development for secure communication. She was an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) in 1998. She joined the computer science department of the University of California, Davis from 2002 to 2003. She has been a professor at Japan Advanced Institute of Science and Technology (JAIST) since 2007. She has been a professor at Graduate School of Engineering, Osaka University since 2015. Her research interests include the application of number theory into cryptography and information security. She received Young Paper Award of SCIS'93 in 1993, Notable Invention Award of the Science and Technology Agency in 1997, the IPSJ Sakai Special Researcher Award in 2002, the Standardization Contribution Award in 2003, the AWARD for the contribution to CULTURE of SECURITY in 2007, the Director-General of Industrial Science and Technology Policy and EnvironmentBureau Award in 2007, DoCoMo Mobile Science Awards in 2008, Advanced Data Mining and Applications (ADMA 2010) Best Paper Award, Prizes for Science and Technology, the Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology, International Conference on Applications and Technologies in Information Security (ATIS 2016) Best Paper Award, the 16th IEEE Trustocm 2017 Best Paper Award, IEICE milestone certification in 2017, the 14th Asia Joint Conference on Information Security (AsiaJCIS 2019) Best Paper Award, Information Security Applications - 20th International Conference (WISA 2020) Best Paper Gold Award, IEICE Distinguished Educational Practitioners Award in 2020, and IEICE Achievement Award in 2023. She is a member of the International Association for Cryptologic Research, the Institute of Electrical and Electronics Engineers, the Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and the Mathematical Society of Japan.