sDPF-RSA: Utilizing Floating-point Computing Power of GPUs for Massive Digital Signature Computations

Jiankuo Dong*†‡, Fangyu Zheng*†, Niall Emmart§, Jingqiang Lin*†‡, Charles Weems§
*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
†Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
‡School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
§College of Information and Computer Sciences, University of Massachusetts, Amherst, MA 01003-4610, USA

***Abstract*—In financial, electronic and other security-sensitive industries, data centers require various protocols and algorithms to secure massive volumes of transactions. It is well known that digital signature is a computationally expensive task and a potential bottleneck that can restrict overall performance. In this paper, we make the following contributions. First, we propose a novel method called sDPF-RSA to accelerate the core algorithm of RSA, Montgomery multiplication, for Graphics Processing Units (GPUs). The sDPF approach takes advantage of the sign bit to increase the amount of information processed with each double precision floating point value and considerably improves performance. Second, we have comprehensively reviewed and tested the algorithms to ensure they all run in constant time. In particular we improve the standard carry resolution algorithm, introducing two constant time parallel techniques. We thus minimize the potential for timing attacks against GPU based RSA crypto-systems. Finally, we propose a full implementation of RSA, optimized for our GPU-accelerated computing platform to maximize its computing power. With protection against timing attacks, the throughputs of RSA-2048/3072/4096 on an NVIDIA GeForce GTX TITAN Black set a record of 52,747/15,179/6,435 (for signature generation) and 1,237,694/584,083/354,139 (for signature verification with public key 65,537) operations per second with modest latency, outperforming the contemporaneous CPU and many-core processor Xeon Phi by 3.9-11 times.**

***Keywords*-Graphics Processing Unit; RSA; Double Precision Floating-point;**

I. Introduction

The proliferation of the Internet has given rise to financial, electronic commerce and other industries which serve huge-scale users. Used ubiquitously in security-sensitive applications of these industries, digital signatures are fundamental to security and involve public key operations which are far more computationally expensive than symmetric operations, e.g., block ciphers and hash functions.

The prevailing digital signatures are RSA and Elliptic Curve Digital Signature Algorithm (ECDSA) which are both standardized in National Institute of Standards and Technology (NIST) Federal Information Processing Standards (FIPS) Publication 186-4 [1]. Although ECDSA is now recommended due to its lower computational cost, RSA is still widely used in many information systems. The requirements for and volume of digital signature are expanding rapidly, straining the capabilities of existing devices. For example, Alipay set a record by processing up to 256,000 payment transactions per second in "Double-Eleven", the Chinese online "Black-Friday" in 2017 [2]. Assume Alipay simply uses one signature verification to identify the buyer and one signature generation to notarize the deal. Thus in one second, Alipay's data center has to process 256,000 signature generation and equal number of signature verification operations. If such a task is outsourced to NShield Connect XC High (released in 2017 [3]), one of the fastest signature servers, which can process 8,600 RSA-2048 signatures per second, at least $\left\lceil \frac{512,000}{8,600} \right\rceil = 60$ of such servers are required to handle the load, let alone the issue of load balancing.

In this contribution, we propose a top-down methodology for RSA signature generation/verification acceleration, providing high-performance, low-latency and also secure signature generation/verification service for highly-concurrent privacy-sensitive online transactions.

A. Related Work

Many previous works reported RSA implementations in different high-performance processors/co-processors, such as common CPU (OpenSSL [4]), GPU, Intel Xeon Phi (PhiOpenSSL [5]), etc. Among them, inspired by the gaming and Artificial Intelligence (AI) industry, GPUs in particular, have developed rapidly and continuously. With the advent of CUDA [6] and OpenCL [7], it is now feasible for GPUs to speed many general-purpose computational workloads.

Both high-definition 3D graphics processing and deep learning applications require high-speed floating-point processing capabilities. From 2010 to the present, the floating-point computing power of CUDA GPUs has grown over ten-fold, from 1,345/665.6 (Fermi architecture) Giga Floating-

This work was partially supported by National 973 Program of China under Award No. 2014CB340603, National Science Foundation (NSF) under Award No. CCF-1525754, National Natural Science Foundation of China under Award No. 61772518 and National Key R&D Program of China under Award No. 2017YFB0802100. *(Corresponding author: Fangyu Zheng, E-mail: fyzheng@is.ac.cn.)*

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00069

point Operations Per Second (GFLOPS) to 15,000/7,500 (Volta architecture) GFLOPS for single/double-precision floating-point (SPF/DPF) arithmetic. While the GPU vendors are putting more efforts into the floating-point units residing in GPUs, the integer multiplication performance of NVIDIA GPUs has seen much more modest gains since Fermi and Kepler. For example, Maxwell and Pascal do not even provide dedicated device instructions for 32-bit integer multiply and multiply-add [6].

Many previous papers implemented RSA using single-threaded methods [8, 9, 10, 11] or distributed methods [9, 12, 11, 13, 14] but were based on the integer computing power of GPUs. Bernstein et al. made an attempt to utilize CUDA floating-pointing processing power in asymmetric cryptography implementations [15]. However, the result was barely satisfactory. Their later work [16] achieved almost 6.5 times the performance of [15] using integer arithmetic on the same GTX 295 card. In 2014 [17] and 2017 [18], we implemented a floating-point-based approach and achieved very efficient results which outperformed the existing fastest integer-based implementations.

B. Contributions and Paper Organization

The contributions of this paper are threefold:

- First, we propose a state-of-the-art method of GPU-accelerated Montgomery multiplication by fully utilizing the DPF computing power of the GPUs, with the approach called the signed-DPF-RSA (sDPF-RSA) method. This effort is based on our previous work [18], but here we leverage the remaining unused computing resource, the sign bit of the floating-point number, to increase the performance.
- Second, as a fundamental security component of the information system, the cryptographic computation itself must be carefully safe-guarded against various kinds of attacks. In this paper, several methods are used to counter the most common side-channel attack on RSA, the timing attack [19], which seriously endangers the security of RSA implementations but is not considered by previous GPU-based implementations. In particular, we propose two approaches to carry resolution: *carry-holding* and *carry-predicting* which run in constant time, and have an additional benefit of improving the performance over traditional ripple carry resolution.
- Third, we have a full implementation of RSA which is optimized for our GPU-accelerated computing platform and can be tuned for either maximum throughput at reasonable latency or minimum latency with reasonable throughput. Our experiments show that when tuned for maximum throughput, our solution outperforms other GPU implementations and contemporaneous CPU and many-core Xeon Phi systems by a large margin. We also perform statistical tests to show that, with high likelihood, the running time is independent of the bit pattern of the decryption key and ciphertext and is safe from timing attacks. The high performance and security shows that GPU-based RSA is an excellent candidate for digital signatures.

The rest of our paper is organized as follows. Section II presents background material. Section III describes our proposed sDPF-based algorithms (conversion algorithms, Montgomery multiplication, and RSA) in detail. Section IV analyses the performance of proposed algorithm and compares it with previous works. Section V concludes the paper.

II. Preliminaries

This section begins with some basic theories of RSA and Montgomery multiplication. Then we introduce the target GPUs and CUDA floating-point arithmetic.

A. RSA and Montgomery Multiplication

RSA [20], an algorithm widely used for digital signature and asymmetric encryption, whose core operation is modular exponentiation, is very computationally expensive. Two different methods are respectively to accelerate the RSA signature generation and verification.

For signature generation, Chinese Remainder Theorem (CRT) [21] is widely used to promote the efficiency. Instead of calculating a $2n$-bit modular exponentiation directly, two n-bit modular exponentiations (Equation (a) & (b)) and the Mixed-Radix Conversion (MRC) algorithm [22] (Equation (c)) can be subsequently performed to conduct the RSA signature generation:

$$P_1 = C^{d \mod (p-1)} \mod p; \quad (a)$$

$$P_2 = C^{d \mod (q-1)} \mod q; \quad (b)$$

$$P = P_2 + [(P_1 - P_2) \cdot (q^{-1} \mod p) \mod p] \cdot q, \quad (c)$$

where p and q are n-bit prime numbers chosen in private key generation ($M = p \times q$). Compared with calculating $2n$-bit modular exponentiations directly, the CRT technology can reduce the computational cost by 75% [21].

For signature verification, the solution is to choose an exponent (i.e., public key) with as few '1's in its binary representation as possible. The smallest exponent that is considered secure now is $2^{16}+1$ (65,537) [1, §B.3.1]. Thus the exponentiation calculation requires only 17 modular multiplications.

Even with a small public exponent and the CRT trick, the bottleneck restricting the overall performance of RSA lies in the modular multiplication. In 1985 Peter L. Montgomery proposed an algorithm [23] to remove the costly division operation from the modular reduction, well known as Montgomery multiplication. Let $\bar{A} = AR(mod\ M), \bar{B} = BR(mod\ M)$ be the Montgomery form of A, B modulo

M, where R and M are co-prime and $M \leq R$. Montgomery multiplication defines the multiplication of two numbers that are in Montgomery form, $MonMul(\bar{A}, \bar{B}) = \bar{A}\bar{B}R^{-1}(mod\ M)$. Even though the algorithm works for any R which is relatively prime to M, it is more useful when R is taken to be a power of 2, which leads to a fast division by R.

B. Timing Attack on RSA

In 1995, Kocher described a new attack on RSA [19]: if the attacker is able to measure the private key times for several known ciphertexts, he/she can deduce the private key quickly. However, for many GPU-based RSA implementations, the focus is mainly on performance, meanwhile potential security risks exist which may leak the information about the key and plaintext. For GPU-based implementations, timing attack is one of the most dangerous ones, for example [19, 24, 25]. A common practice to thwart this attack is to ensure that the computation takes a constant amount of time for every ciphertext and key, which however always significantly reduces performance.

C. Target GPU and Floating-point Arithmetic in CUDA GPUs

Our target GPU is the GTX TITAN Black, a CUDA-compatible GPU with Compute Capability (CC) 3.5, which contains 15 streaming multiprocessors (SMs). 32 threads are grouped together as a *warp* and multiple warps are assigned to each SM. Instructions are dispatched concurrently to all 32 threads in the warp. Warps are further grouped into *blocks* and *grids*. Blocks are assigned to individual SMs and grids are spread across all the SMs on a GPU device. See [6] for a complete description of the GPU architecture and programming model. Table I gives the throughputs of the native arithmetic instructions (per SM per clock) on the GTX TITAN Black. It is the high throughput of DPF arithmetic instructions that makes the sDPF-RSA approach attractive.

Table I
THROUGHPUT OF NATIVE ARITHMETIC INSTRUCTIONS (NUMBER OF OPERATIONS PER CLOCK CYCLE PER SM)

	SPF	DPF	32-bit add	32-bit multiply
Throughput(/SM/CLK)	192	64	160	32 (16)*

* With CC 3.5, to compute the full product (32-bits $\times$ 32-bits) $\rightarrow$ 64-bits requires two instructions to compute the lower and upper halves, thus the full-product throughput is 16.

Floating-point arithmetic instructions in CUDA GPUs comply with 754-2008 IEEE Standard for Floating-Point Arithmetic [26]. Both 32-bit binary SPF and 64-bit binary DPF are supported by CC 3.5 GPUs. SPF and DPF can exactly represent a 24-bit and 53-bit integer respectively. The GTX TITAN Black also supports SPF and DPF fused-multiply-and-add (FMA) instructions, which are executed in a single instruction dispatch. FMA computes the product and sum to infinite precision and then does a single rounding step to produce the result.

D. Data Sharing Functions in Target GPU

NVIDIA GPUs with compute capability 3.0 and higher support high performance inter-thread communication within a warp via *shuffle* instruction and `__ballot` voting function.

- The instruction *shuffle* uses a full cross-bar switch and allows each thread to exchange a 32-bit value with any other thread within the warp.
- The function `__ballot` evaluates a predicate for all active threads of the warp and returns a 32-bit integer whose N-th bit is set if and only if predicate evaluates to non-zero on the N-th thread of the warp.

Shuffle and ballot operations are used extensively in the algorithms that follow.

III. METHODOLOGY

This section details our implementation of RSA. From a developer's perspective, the implementation can be decomposed into two levels: the high-level routines operate on entire big numbers. These routines would include the RSA algorithm, the MRC algorithm and the modular exponentiation algorithm. The low-level routines deal with the big number representations and the implementation of the Montgomery multiplication. The high-level routines are well known and in the interest of space we give only a brief overview. The low-level representations and routines are the main focus of this section. Figure 2 shows the overall architecture of sDPF-RSA.

A. High Level Routines

1) RSA signature verification and modular exponentiation: Modular exponentiation is the fundamental component of RSA. With the binary square-and-multiply method, the expected number of modular multiplications is $3n/2$ for n-bit modular exponentiation. The number can be reduced with fixed window method for modular exponentiation given by Knuth [27] that scans multiple bits, instead of one bit of the exponent. Jang et al. [12] and Yang et al. [13] used sliding window technology Constant Length Nonzero Windows (CLNW) [28] to reduce the number of modular multiplication further. This method has two drawbacks. First, sliding window techniques are susceptible to timing attacks because the number of multiply steps will depend on the bit pattern of the exponent. Second, it is not suitable for our design, in which an entire warp may contain more than one modular multiplication and each modular multiplication has a different exponentiation which can lead to warp divergence and decreased overall performance. Thus, we use a fixed window method, where the number of square and multiply steps are constant regardless of exponent. The modular

exponentiation routine is built on a sub-routine, modular multiplication (Montgomery multiplication).

As aforementioned in Section II-A, RSA signature verification (i.e., public encryption) can directly utilize modular exponentiation routine with small exponent ($2^{16}+1$). Note that the exponent is fixed and small, we can directly use the binary square-and-multiply method, which requires only 16 modular squaring operations and one modular multiplication.

2) RSA signature generation: Accelerated by CRT technique, RSA signature generation (i.e., private decryption) is more complex than signature verification, which additionally requires a MRC computation. MRC computation includes a modular multiplication, a modular subtraction and a multiply-add. Modular multiplication can be implemented via our proposed method described in Section III-B, and for the modular subtraction and the normal multiply-add, we multiplex the thread resources of modular multiplication following a similar sDPF-based approach as shown in Figure 1.

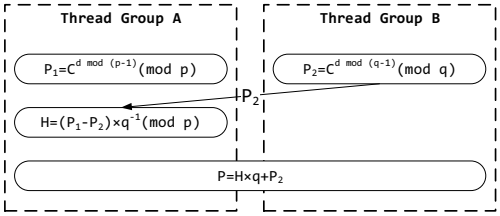

Figure 1. Diagram for RSA signature generation

B. Low Level Routine: Montgomery Multiplication

In this section, we propose a Montgomery multiplication based on a novel big number representation using signed DPFs. Our approach is based on Koç et al.'s [29] Coarsely Integrated Operand Scanning (CIOS) method for computing $MonMul(A,B)$. Koç's method works as follows: initialize S to zero, then iterate over the limbs of B, updating $S=(S+A\cdot b[i]+q\cdot M)/2^w$, where w is the limb size and q is the smallest value such that $S+A\cdot B[i]+q\cdot M$ is evenly divisible by 2^w. After the iteration, if $S>M$ then a final correction step is needed. Our approach differs from Koç's in three important aspects: first, as already mentioned, we use a novel redundant big number representation; second, we accumulate the carries locally during the iteration and resolve them at the end; third, instead of using a sequential approach, we parallelize the big number computations across a group of threads within a warp. Table II defines the variables which will be used in the following sections.

C. Number Representation

In a traditional setting, big numbers are represented as a sequence of 32 or 64-bit unsigned integer limbs. In our approach, we store integer limbs in a sequence of DPF values. We use three (related) representations in different parts of the computation. For each representation we always have $A=\sum_{i=0}^{l-1} a_i\cdot 2^{wi}$, however, the bounds on the a_i terms vary as follows:

- *Unique (non-redundant) unsigned format:* is a traditional w-bit representation, where $0\le a_i<2^w$. It is used to represent the input and output of the RSA computation.
- *Redundant sDPF format:* a_i contains at most **53** significant bits. It is used to accommodate the output of the FMA instruction.
- *Simplified sDPF format:* $-2^{w-1}-1\le a_i\le 2^{w-1}+1$. Note: this is also a redudant format, but with much smaller ranges.

Table II
SYMBOL EXPLANATION

Symbol	Explanation
n	The bit length of the modulus
w	Number of significant bits in each DPF limb
l	Number of DPF limbs of the DPF modulus, where $l=\lceil\frac{n+1}{w}\rceil$
r	Number of threads per Montgomery multiplication (r must divide 32)
k	Thread ID where $0\le k\le r-1$
t	Number of DPF limbs per thread, where $t=\lceil\frac{l}{r}\rceil$
v	The window size of fixed window exponentiation algorithm

Since our limbs are DPF values, we perform all computations on limbs with DPF arithmetic. A key point is that the intermediate values during the computation must never exceed $\pm 2^{53}$, the largest integer value that can be exactly represented. If they do, then round-off could occur, leading to incorrect results. Switching from the unsigned DPF representation used in our earlier work [18] to the proposed signed DPF representation, will allow us to accumulate more limb products in a DPF value without overflowing the $\pm 2^{53}$ limit, and thus it takes fewer limbs to represent the same big number. Table III allows us to estimate the number of FMA instructions saved by using the more compact representation as 8% at 2048 bits to 12% at 1024 bits.

Table III
NUMBER OF FMA PER MONMUL: SDPF VS. DPF ($r=4$)

n length (bit)		1024	1536	2048
DPF w		23	22	22
sDPF w		24	23	23
DPF l		45	70	94
sDPF l		43	67	90
DPF t		12	18	24
sDPF t		11	17	23
# of FMA	DPF	4,320	10,080	18,048
	sDPF	3,784	9,112	16,560

Unfortunately, the GPU does not have enough register resources to support a modular multiplication instance in

each thread. To work around this problem, we use a distributed representation, where a group of r threads handles each modular multiplication instance. The limbs of A, B, and M are divided into r contiguous chunks and distributed to the r threads.

Finally, we note that the degree of parallelism r (which must divide the warp size of 32) can be tuned to offer either high throughput with adequate latency or low latency with adequate throughput.

D. Processing Phases

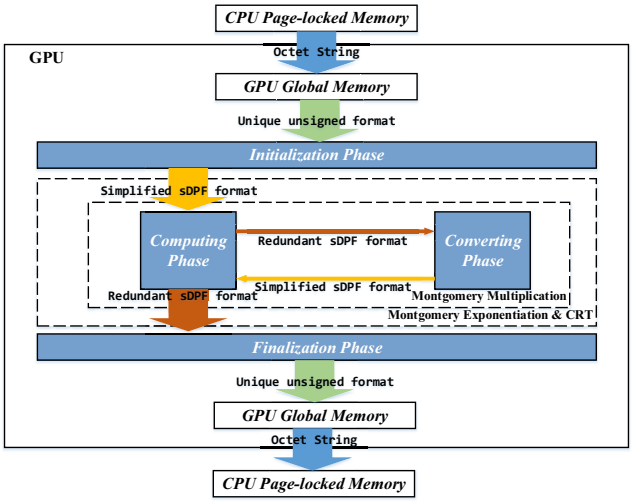

Figure 2. Overall Architecture of sDPF-RSA

An overview of the processing phases and associated big number representations are given in Figure 2. The processing occurs in four phases: *Initialization Phase*, *Computing Phase*, *Converting Phase*, and *Finalization Phase*. These are shown as shaded blocks in the diagram and are described in detail below.

1) Initialization Phase: In this phase, the data is transferred from the CPU in a packed binary format (octet string). The first step is that octet strings are divided into limbs of w-bits, and the limbs are distributed to the r threads. At this point, the big numbers are in the unique unsigned format. Next, Algorithm 1 is run, which converts the big numbers to simplified sDPF format. Algorithm 1 works as follows. First each thread turns its l limbs into simplified sDPF format in Steps 2-9. Next, thread $k-1$ transmits its $carry$ to thread k (where $k \neq 0$) and adds it to a_0. Note that, at this point, $(-2^{w-1}+1) \leq a_0 \leq 2^{w-1}$, a_0 is also in the range of simplified sDPF format, so no further carry propagation is required. And the *if-else* (Line 4-8) can be replaced with a constant-time and efficient approach described in Section III-E.

Algorithm 1 Data Type Conversion in *Initialization Phase*

Input:
The n-bit big number A in the *unique unsigned format*, $A = \sum_{i=0}^{l-1} a_i 2^{wi}$, where $0 \leq a_i < 2^w$

Output:
The *simplified sDPF format* representation of A, $\widehat{A} = \sum_{i=0}^{l-1} \widehat{a_i} 2^{wi}$, where $-2^{w-1}+1 \leq \widehat{a_i} \leq 2^{w-1}$

1: $carry = 0$
2: **for** $i = 0$ *to* $l-1$ **do**
3: $\quad a_i = a_i + carry$
4: $\quad$ **if** $a_i < 2^{w-1}$ **then**
5: $\quad\quad carry = 0$, $\widehat{a_i} = a_i$
6: $\quad$ **else**
7: $\quad\quad carry = 1$, $\widehat{a_i} = a_i - 2^w$
8: $\quad$ **end if**
9: **end for**
10: Obtain $carry$ from previous thread using *shuffle*
11: $\widehat{a_0} = \widehat{a_0} + carry$

We note that the conversion from A to $\widehat{A}$ (in Algorithm 1) does not change the big number value, i.e.,

$$\sum_{i=0}^{l-1} a_i \cdot 2^{wi} = \sum_{i=0}^{l-1} \widehat{a_i} \cdot 2^{wi}$$

whenever we subtract 2^w from $\widehat{a_i}$, we add 1 to $\widehat{a_{i+1}}$, thus preserving the total value. Since we haven't changed the basic representation or the total value, the classic arithmetic algorithms will produce the same result value whether operating on A or $\widehat{A}$. The only thing that will be different will be the result representation.

2) Computing Phase: With the big numbers in simplified sDPF format, we are ready to compute a Montgomery product. The multi-threaded computation is shown in Figure 3. This computation is analogous to Koç et al.'s CIOS method as described above and applied to our big number representation. We note that the maximum value of each column sum, $|S[i]| \leq 2l \cdot (2^{w-1}+1)^2 + 2^{53-w}$, which must be less than 2^{53} to avoid round-off. Koç's CIOS method requires a final correction step, however, in 1995, Orup [30] discovered that relaxing the restriction of input and output from $[0, M)$ to $[0, 2M)$, meant the correction step could be completely removed. Detailed algorithm is shown in Algorithm 2.

This solution is ideal for our purposes, since the modular exponentiation sizes of interest, 1024-bit, 1536-bit, and 2048-bit, are not evenly divisible by their corresponding w, thus, we always have at least one "free" bit in the most significant limb. This means we can accommodate Orup's larger values without requiring additional limbs. Removing the correction step is fortuitous, since the correction step would require a comparison, which is very difficult to perform in a redundant representation. Further, from the per-

Algorithm 2 Revised Montgomery Multiplication (CIOS Method) According to Orup's Method

Input:

$M > 2$ with gcd(M,2)=1, , positive integers l, w such that $2^{wl} > 4M$

$M' = -M^{-1} \mod 2^w$, $R^{-1} = (2^{wl})^{-1} \mod M$

Integer multiplicand A, where $0 \leq A < 2M$

Integer multiplier B, where $B = \sum_{i=0}^{l-1} b_i 2^{wi}$, $0 \leq b_i < 2^w$ and $0 \leq B < 2M$

Output:

An integer S such that $S = ABR^{-1} \mod 2M$

1: $S = 0$
2: **for** $i = 0$ *to* $l - 1$ **do**
3: $S = S + A \times b_i$
4: $q_i = ((S \mod 2^w) \times M') \mod 2^w$
5: $S = (S + M \times q_i)/2^w$
6: **end for**

spective of security, Orup's method makes the Montgomery multiplication run in constant time, which can prevent timing attacks to the correction step [19].

3) Converting Phase: At the end of Computing Phase, S is in redundant sDPF format, i.e., $-2^{53} < a_i < 2^{53}$. Before we can perform another Montgomery multiplication, we must bring the representation back to a simplified sDPF format. In a sequential implementation, this would be easy to do: initialize the $carry$ to zero, then $S[i]$ adds the $carry$ and holds the least significant w bits of the sum and propagates the most significant $(53 - w)$ bits as the new $carry$ to $S[i + 1]$. This would reduce the big number to unique unsigned format, then Algorithm 1 could be used to convert to simplified sDPF format.

Algorithm 3 Converting Phase with carry-holding method

Input:

$s[0 : t - 1]$: Redundant-format sub-result and $carry$, where $s[0 : t - 1] = S[tk : tk + t - 1]$;

Output:

$s[0 : t - 1]$: where $s[0 : t - 1] = S[tk : tk + t - 1]$, $s_i \in [0, 2^w - 1]$, and $carry$;

1: **for** $j = t - 1 - (\lceil \frac{53}{w} \rceil - 1)$ to $t - 1$ **do**
2: $s[j] = s[j] + carry$
3: $carry = s[j] >> w$
4: $s[j] = s[j]$ & $(1 << w - 1)$
5: **end for**
6: Obtain $carry$ from previous thread using *shuffle*
7: **for** $j = 0$ to $t - 1$ **do**
8: $s[j] = s[j] + carry$
9: $carry = s[j] >> w$
10: $s[j] = s[j]$ & $(1 << w - 1)$
11: **if** $s[j] >= 2^{w-1}$ **then**
12: $carry = carry + 1$, $s[j] = s[j] - 2^w$
13: **end if**
14: **end for**
//At this point, for any i, $s[i] \in [-2^{w-1}, 2^{w-1} - 1]$; $carry$ may be -1, 0, 1, or 2
15: Obtain $carry$ from previous thread using *shuffle*
16: $s[0] = s[0] + carry$
//Finally, $s[0] \in [-2^{w-1} - 1, 2^{w-1} + 1]$

But in a distributed implementation it becomes more complex. In the past, several authors [12, 17, 18] have explored a ripple-carry approach (a.k.a., lazy-carry propagation), where the carry is resolved locally within the thread. Any carry outs are passed to the next thread up as a carry in and resolved

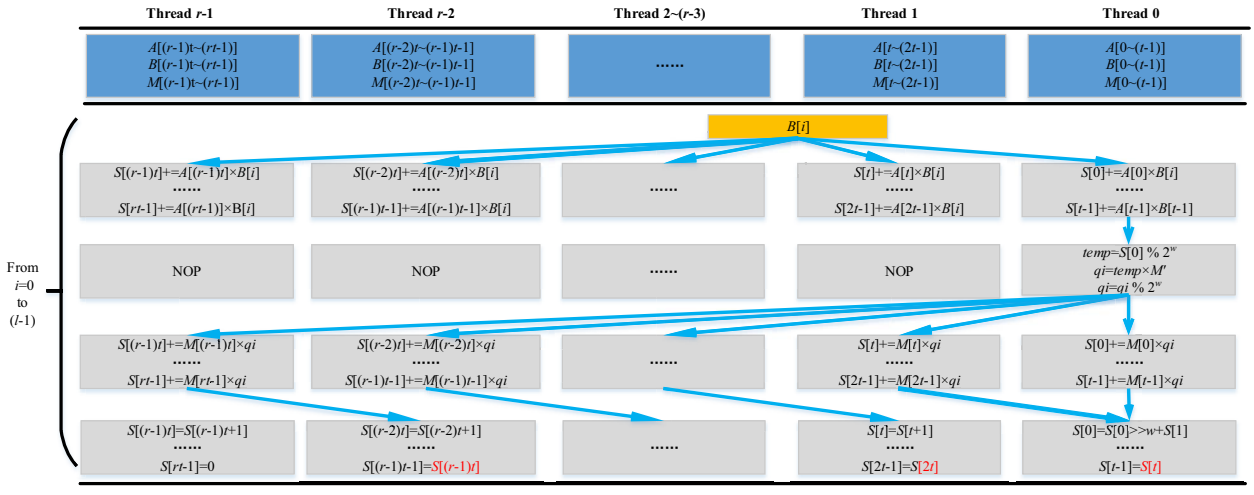

Figure 3. *sDPF-based Computing Phase* of Montgomery Multiplication

again. This process repeats until there are no more carry outs. The ripple-carry approach generally terminates after two iterations. However, it is possible for a carry to ripple from the least significant thread all the way to the most significant thread. Thus, the method is not constant-time and may be susceptible to timing attacks.

Here we propose a new method, which we call *carry-holding* and is presented in Algorithm 3. We take advantage of the fact that simplified sDPF format is itself a redundant format. Thus, a thread can absorb a carry in of -1, 0, 1, or 2 from a lower thread without ever generating a carry out. Therefore, at most two carry resolution iterations are required, and it turns out, the first resolution need only process the most significant $\lceil \frac{53}{w} \rceil - 1$ limbs. For detailed limits on the limb and carry values, please see the comments in the algorithm. Note that, the *if* control flow (Line 11-13) can be also replaced with a constant-time and efficient approach described in Section III-E.

The carry-holding approach is a significant improvement. Compared with the traditional ripple-carry approach, benefits are threefold: firstly, ripple-carry approach may impact on every limb in every iteration, thus the unsigned-to-signed conversion (Line 11-13 of Algorithm 3) cannot be applied until ripple-carry propagation finishes; meanwhile, they can be combined in carry-holding method. Secondly, carry-holding method largely promotes the performance with less computation and carry propagation, i.e., substituting a series of expensive loops with a simple DPF addition (Line 16 of Algorithm 3). Last but not the least, this method is constant-time, which is timing-attack proof.

4) Finalization Phase: This is run once at the end of the processing and is used to convert the redundant sDPF format to a unique (non-redundant) format (after which the numbers are packet into octet strings for transmission to the CPU). The finalization phase first runs a variant of Algorithm 3 with lines 11-13 removed. Without these lines, Algorithm 3 produces a carry out from each thread of -1, 0, or 1. Algorithm 4 is parallel approach to resolve all the carries in constant time, called *carry-predicting* method.

The basic idea of carry-predicting method is as follows: firstly, utilizing `__ballot` voting function to detect whether the thread i ($i \in [0, 31]$) generates carry (g_i) and is potential to carry to next thread (p_i); note that `__ballot` would return a 32-bit integer $G = (g_{31}, g_{30}, \ldots, g_0)$ and $P = (p_{31}, p_{30}, \ldots, p_0)$. Then it "predicts" carry via

$$C = (G << 1 + P) \oplus P \tag{1}$$

of which each bit of C represents the carry of the corresponding thread, and then each thread simply adds its own carry. Figure 4 explains the process by a simple example.

Note that, we should separately process the negative and positive carry. Algorithm 4 details the carry-predicting method.

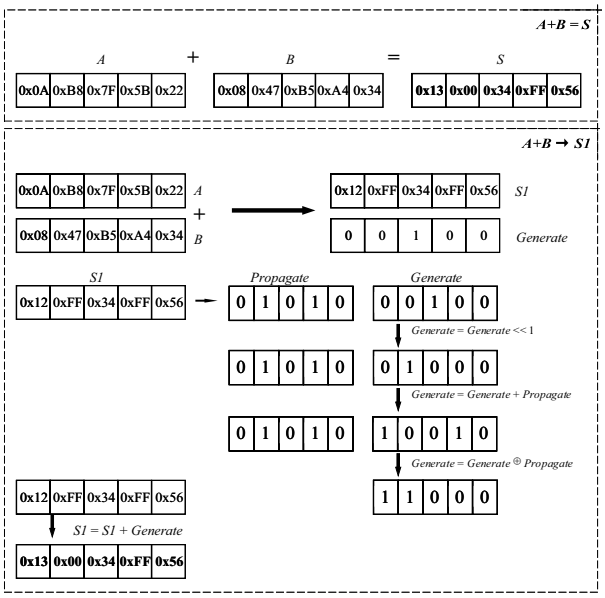

Figure 4. A simple example of carry-predicting method

Algorithm 4 *Finalization Phase* with Carry-predicting Method

Input:

$s[0 : t-1]$: Redundant-format sub-result and $carry$, where $s[0 : t-1] = S[tk : tk + t - 1]$;

Output:

$s[0 : t-1]$: where $s[0 : t-1] = S[tk : tk + t - 1]$, $s_i \in [0, 2^w - 1]$, and $carry$;

1: Process $s[0 : t-1]$ following Line 1-10 and 14 of Algorithm 3 //At this point, $carry$ may be -1, 0, 1
//Process negative carry
2: p_i = if all limbs in current thread are zero
3: P = `__ballot`(p_i & $carry = 0$);
4: G = `__ballot`($carry = -1$);
5: Compute $carry_{neg}$ using Equation (1)
6: $carry_{neg}$=$carry_{neg}$&!($p_i = 1$ & $carry = 1$)
7: $s[0 : t-1] = s[0 : t-1] - carry_{neg}$
//Process positive carry
8: P = `__ballot`(if all limbs in current thread are $2^w - 1$);
9: G = `__ballot`($carry = 1$);
10: Compute $carry_{pos}$ using Equation (1)
11: $s[0 : t-1] = s[0 : t-1] + carry_{pos}$

E. Implementation Tricks

1) Bit manipulation: In processing phases, sometimes we have to manually manipulate the bits of DPF limbs which is inefficient when directly using DPF instructions, such as the shifts and `AND`s in Algorithm 1 and 3. Our general approach is, converting the DPF limbs into 64-bit integers, then manipulating bits using native bit-wise instructions and finally converting them back. But there are two special and

noteworthy situations.

Firstly, Line 2-9 of Algorithm 1 and Line 11-13 of Algorithm 3 can be efficiently processed in constant time by followed steps,

```
add.cc.u32    a, a, carry_input;
addc.u32      carry_output, 0, 0;
xor.b32       a, top_mask;
```

where integer addition-with-carry instruction `addc.cc` is applied (`top_mask` indicates a 32-bit mask leading with $33 - w$ ones and followed by zeros).

Secondly, in Computing Phase, we can quickly compute right shift $S[0] >> w$ simply using one DPF division operation $S[0]/2^w$ since the CIOS method can ensure 2^w divides $S[0]$.

2) Page-lock memory: The page-locked memory on the CPU (obtained with `cudaMallocHost`) is used to accelerate data transfer, which can be accessed directly by the device thus offers higher bandwidth than pageable memory (obtained with `malloc`) [6], especially for signature verification (the overall latency decreases by up to 18%).

IV. Performance Evaluation and Related Work Comparison

A. Performance Evaluation

This section discusses the implementation performance and summarizes the results for the proposed algorithm with comparisons to other GPU, CPU and Xeon Phi implementations. We build a signature server prototype with the following hardware and software configuration which is listed in Table IV.

Table IV
Experiment Configuration

CPU	Intel Xeon CPU E5-2690 at 2.90GHz
GPU	GeForce GTX TITAN Black
OS	Ubuntu 16.04
Tool Chain	CUDA 8.0
Test Key Source	OpenSSL 1.0.2g *RSA_generate_key*() API
Test Message Source	Linux pseudorandom number generator "/dev/urandom"

The RSA-2048/3072/4096 signature generation/verification algorithms are implemented respectively followed sDPF-based Montgomery multiplication algorithms and RSA implementation in Section III. There are two metrics for evaluating the performance of RSA accelerator.

- *Throughput*: the numbers of RSA signature generation/verification completed in one period (e.g., 1s) divided by the period;
- *Latency*: the waiting interval from requesting RSA computing to getting the computed results.

Several configuration parameters may affect the performance of the kernel, including:

- *Batch Size*: the number of RSA operation per GPU kernel launch.
- *Threads/RSA*: the number of threads assigned for each RSA signature generation/verification.
- *Threads/Block*: the number of threads in each GPU block.
- *Regs/Thread*: the maximum number of registers assigned for each CUDA thread; both *Regs/Thread* and *Threads/Block* $\times$ *Regs/Thread* should be restricted within the GPU hardware limitation (255 and 65536).

Since throughput is the most important metric for our work, we choose the maximum *Batch Size* supported by the GPU hardware assuming 128 registers per thread. In Figure 5, the performance is represented as 128_Throughput and 128_Latency. Performance with 255 registers per thread, whose symbols are 255_Latency and 255_Throughput, is also provided for comparison.

Table V
Performance of RSA Signature Verification (with public key 65,537)

	Threads/RSA	Batch Size	Throughput (ops/s)	Latency (ms)
RSA-2048	8	15×64	**1,237,694**	0.78
	16	15×32	928,433	**0.52**
RSA-3072	8	15×64	**584,083**	1.66
	16	15×32	575,953	**0.83**
RSA-4096	8	15×64	120,108	7.99
	16	15×32	**354,139**	**1.35**

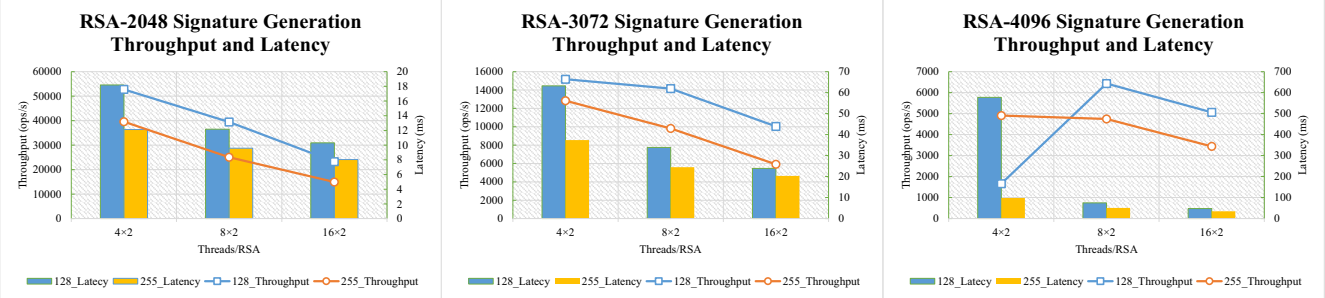

Figure 5. Performance of RSA Signature Generation

B. Security Evaluation

Timing attack is a critical issue for a practical RSA server and must be addressed in every component of the system. In this section, we provide a summary of the counter-measures employed to ensure constant-time execution and the relevant section:

1) Constant-time modular exponentiation approach (Fixed window method) in Section III-A.
2) Correction step removal (with Orup et al.'s approach) in Section III-D;
3) Constant-time carry resolution approaches (the proposed carry-holding and carry-predicting methods) in Section III-D;
4) Control flow improvement in Section III-E;

To verify the efficiency of the countermeasures, we use one-way analysis of variance (ANOVA) to evaluate whether the running time is independent of the keys and ciphertext. One way ANOVA is a statistical technique that compares expected values of two or more distributions [31]. We generate 1000 different RSA keys using the OpenSSL library. And for each key, we collect 1000 samples of the corresponding latency. Without loss of generality, we measure the latency of one-block kernel, which ensures the GPU does not become too hot and the clock rate remains as stable as possible.

We conduct one-way ANOVA for the 1000 groups of 1000 samples via Microsoft Excel ANOVA tools, whose RSA-2048 results is shown in Table VI. We can find that P-value is 97.1%, concluding that there is a strong evidence (97.1%) that the expected values in the 1000 groups are the same, in other words, the latency distributions are independent of the key and cipher text.

Table VI
ONE-WAY ANOVA FOR RSA-2048 SIGNATURE GENERATION LATENCY

V.S.[*]	SS	df	MS	F	P-value	F_{crit}(95%)
B.G.[*]	130.3589	999	0.130	0.917	0.971	0.928
W.G.[*]	142183.2	999000	0.142			
Total	142313.6	999999				

[*] V.S.: Variation Source, B.G.: Between Groups, W.G.: Within Groups

Another attack on RSA (using CRT) that has been reported is the fault attack (see for example Boneh-DeMillo-Lipton attack [32]). Thanks to our efficient implementation of signature verification, this attack can be effectively and efficiently thwarted by verifying the signature before returning. This extra verification step only causes 1.9%-4.8% performance loss. Considering that the hardware fault attack for GPUs is not reported by now, this countermeasure may be used in some high-security-level scenarios.

C. Related Work Comparison

1) vs. CPU and Xeon Phi: This section compares our work with related work based on CPU (OpenSSL [4]) and Xeon Phi (PhiOpenSSL [5]). Figure 6 demonstrates the throughput and latency comparison (we evaluate OpenSSL via its console command by fully occupying the 24 hardware threads of Intel Xeon Processor E5-2697 v2).

As shown in Figure 6, we achieve up to 11.03 times throughput of non-timing-attack-proof PhiOpenSSL with lower latency. Our implementation also outperforms OpenSSL by a factor of 3.94 and 6.10 on RSA-4096 and RSA-2048 respectively. OpenSSL has an advantage in latency, however, for most applications, the throughput is the most important metric, assuming the latency is reasonable (e.g., less than 100 ms). It is also worth noting, the Xeon E5-2697 v2 (Q3'13) and Xeon Phi Knights Corner ((Q2'13)) were released at almost the same time with our GPU platform (Q1'14), while the price of our GPU platform ($1000) is much lower than both of them (about $2000 to $3000).

2) vs. GPU works: Table VII presents the performance of our sDPF-based RSA implementation and compares it to earlier GPU efforts.

On the Kepler architecture, our sDPF-RSA outperforms most of the previous works [11, 13, 18] on both throughput and latency, even considering the differences in cards. Emmart and Weems achieved a higher throughput for RSA-2048 on the comparable 780 Ti, but the batch sizes (requiring tens of thousands of concurrent requests) and latency are too high for practical applications. Compared with our previous work [18], we improve the throughput by up to 10%, reduce the latency, and resolve the issue of timing attacks.

Comparing our proposed solution to Jang et al. [12] is a bit trickier due to the differences in architecture (Fermi vs. Kepler). However, we can estimate the expected performance of Jang's solution on a GTX Titan Black by scaling for the number of SMs, the integer multiplication throughput per cycle and the CUDA core clock rate. We find that Jang's solution's performance should be roughly the same on a GTX 580 as on a GTX Titan Black, and our proposed method provides several times the throughput performance with a comparable latency. We believe the performance gap is mainly due to two issues: the lower throughput of integer multiplications and Jang's use of 32 threads for each RSA, which greatly increases the burden of thread communication and synchronization.

Pan et al. [33] implement an ECDSA-based [1] signature server based on GTX 780 Ti, whose ECDSA-P256 signature generation/verification respectively reaches 8.71 (with 0.42 ms latency) and 0.929 (with 28.52 ms latency) million operations per second (MOPS) without explicit protection for timing-attack. Their huge performance advantage in signature generation mainly comes from the underlying cryptographic algorithms, which has much lower computational cost [8], and allows off-line pre-computing. But for signature verification, we still outperform it in both throughput (1.24 MOPS vs. 0.929 MOPS) and latency (0.78 ms vs. 28.52 ms).

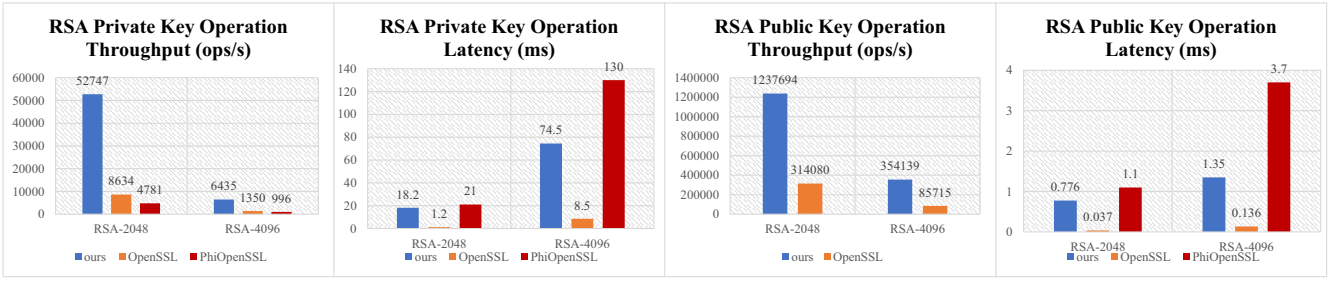

Figure 6. vs. OpenSSL and PhiOpenSSL

Table VII
VS. GPU WORKS

	Jang et al. [12]	Emmart et al. [11]	Yang et al. [13]	Dong et al. [18]	Ours	
CUDA platform	GTX 580	GTX780Ti	GT 750m	GTX TITAN	GTX TITAN	GTX TITAN Black
Architecture	Fermi	Kepler	Kepler	Kepler	Kepler	Kepler
# of SMs	16	15	2	14	14	15
Shader clock (GHz)	1.544	0.876	0.967	0.837	0.837	0.889
Secure at all key sizes	No	No	No	No	Yes	Yes
RSA-2048 (ops/s)	12,044	62,365	5,244	42,211	46,715	52,747
RSA-2048 (ms)	13.83	60.07	195.27	21.22	19.18	18.20
RSA-3072 (ops/s)	-	-	-	12,151	13,385	15,179
RSA-4096 (ops/s)	-	5,257	-	5,790	6,007	6,435

Finally, we do not list the work of [14] in Table VII. Their solution is latency-oriented and unfortunately, they do not report throughput. Our estimate is that it can offer an order of magnitude lower latency than ours but its throughput is roughly an order of magnitude lower than ours.

In conclusion, our results show that we have both improved the performance, practicality, and security over prior efforts. We believe our results have shown all the components to necessary to build a practical GPU based RSA digital signature server.

V. CONCLUSION

In this paper, we implement a novel, systematic and secure implementation of RSA signature scheme. Our primary motivation is maximizing the power of GPUs for RSA signature scheme and making it ready-to-use for practical usages. To attain this end, a novel method called sDPF-RSA is proposed to push the core algorithm to the limit, and a systematic RSA implementation is customized and optimized. In particular, we use various methods to prevent from the dangerous timing attack, which is not considered by previous GPU-based implementations. Experiments have shown that we achieve significantly higher performance than existent prototypes or products. In fact, some of our contributions have been already applied to the commercial GPU-based HSM (Hardware Security Module) [34].

Our future work will focus on fast and secure implementations of prevailing signature schemes, include ECDSA, Edwards-curve Digital Signature Algorithm (EdDSA) [35], etc.

REFERENCES

[1] U. D. of Commerce, N. I. of Standards, and Technology, "Digital signature standard (DSS)," http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013.

[2] C. Team. (2017). [Online]. Available: https://www.chinainternetwatch.com/22791/double-11-2017

[3] T. e Security. (2017) nShield Connect Data Sheet. [Online]. Available: https://www.thalesesecurity.com/products/general-purpose-hsms/nshield-connect

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

[5] S. Yao and D. Yu, "PhiOpenSSL: Using the Xeon Phi coprocessor for efficient cryptographic calculations," in *Parallel and Distributed Processing Symposium*, 2017, pp. 565–574.

[6] NVIDIA, "CUDA C programming guide 9.0," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[8] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Cryptographic Hardware and Embedded Systems–CHES 2008*. Springer, 2008, pp. 79–99.

[9] O. Harrison and J. Waldron, "Efficient acceleration of asymmetric cryptography on graphics hardware," in *Progress in Cryptology–AFRICACRYPT*

2009. Springer, 2009, pp. 350–367.

[10] S. Neves and F. Araujo, "On the performance of GPU public-key cryptography," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 133–140.

[11] N. Emmart and C. Weems, "Pushing the performance envelope of modular exponentiation across multiple generations of GPUs," in *2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015. Proceedings*, 2015.

[12] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: cheap SSL acceleration with commodity processors," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 1–1.

[13] Y. Yang, "Accelerating rsa with fine-grained parallelism using GPU," in *In: Lopez J., Wu Y. (eds) Information Security Practice and Experience. Lecture Notes in Computer Science, vol 9065. Springer, Cham*, 2015.

[14] N. Cruz-Cortés, E. Ochoa-Jiménez, L. Rivera-Zamarripa, and F. Rodríguez-Henríquez, "A GPU parallel implementation of the RSA private operation," in *Latin American High Performance Computing Conference*. Springer, 2016, pp. 188–203.

[15] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang, "ECM on graphics cards," in *Advances in Cryptology-EUROCRYPT 2009*. Springer, 2009, pp. 483–501.

[16] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang, "The billion-mulmod-per-second PC," in *Workshop record of SHARCS*, vol. 9, 2009, pp. 131–144.

[17] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUs for RSA," in *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, 2014, pp. 198–215.

[18] J. Dong, F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Utilizing the double-precision floating-point computing power of GPUs for RSA acceleration," *Security and Communication Networks*, vol. 2017, 2017.

[19] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[20] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[21] J.-J. Quisquater and C. Couvreur, "Fast decipherment algorithm for RSA public-key cryptosystem," *Electronics letters*, vol. 18, no. 21, pp. 905–907, 1982.

[22] C. K. Koç, "High-speed RSA implementation," Technical Report, RSA Laboratories, Tech. Rep., 1994.

[23] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[24] W. Schindler, *A Timing Attack against RSA with the Chinese Remainder Theorem*. Springer Berlin Heidelberg, 2000.

[25] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *IEEE International Symposium on High PERFORMANCE Computer Architecture*, 2016, pp. 394–405.

[26] I. S. Committee *et al.*, "754-2008 IEEE standard for floating-point arithmetic," *IEEE Computer Society Std*, vol. 2008, 2008.

[27] D. E. Knuth, "The art of computer programming: seminumerical algorithms, Vol. 2Addison-Wesley," *Reading, MA*, p. 116, 1981.

[28] C. K. Koç, "Analysis of sliding window techniques for exponentiation," *Computers & Mathematics with Applications*, vol. 30, no. 10, pp. 17–24, 1995.

[29] C. K. Koç, T. Acar, and B. S. Kaliski Jr, "Analyzing and comparing Montgomery multiplication algorithms," *Micro, IEEE*, vol. 16, no. 3, pp. 26–33, 1996.

[30] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*. IEEE, 1995, pp. 193–199.

[31] Wikipedia, "Wikipedia: One-way analysis of variance," https://en.wikipedia.org/wiki/One-way_analysis_of_variance, 2017.

[32] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in CryptologyEUROCRYPT97*. Springer, 1997, pp. 37–51.

[33] W. Pan, F. Zheng, W. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Transactions on Information Forensics and Security*, 2017.

[34] Zanjia and Atsec, "HSM-ZJ2014 FIPS 140-2 Non-Proprietary Security Policy," https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp2692.pdf, 2016.

[35] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (eddsa)," Tech. Rep., 2017.