Side-channel Timing Attack of RSA on a GPU

CHAO LUO, MathWorks, USA

YUNSI FEI and DAVID KAELI, Northeastern University, USA

To increase computation throughput, general purpose Graphics Processing Units (GPUs) have been leveraged to accelerate computationally intensive workloads. GPUs have been used as cryptographic engines, improving encryption/decryption throughput and leveraging the GPU's Single Instruction Multiple Thread (SIMT) model. RSA is a widely used public-key cipher and has been ported onto GPUs for signing and decrypting large files. Although performance has been significantly improved, the security of RSA on GPUs is vulnerable to side-channel timing attacks and is an exposure overlooked in previous studies.

GPUs tend to be naturally resilient to side-channel attacks, given that they execute a large number of concurrent threads, performing many RSA operations on different data in parallel. Given the degree of parallel execution on a GPU, there will be a significant amount of noise introduced into the timing channel given the thousands of concurrent threads executing concurrently.

In this work, we build a timing model to capture the parallel characteristics of an RSA public-key cipher implemented on a GPU. We consider optimizations that include using Montgomery multiplication and sliding-window exponentiation to implement cryptographic operations. Our timing model considers the challenges of parallel execution, complications that do not occur in single-threaded computing platforms. Based on our timing model, we launch successful timing attacks on RSA running on a GPU, extracting the private key of RSA. We also present an effective error detection and correction mechanism. Our results demonstrate that GPU acceleration of RSA is vulnerable to side-channel timing attacks. We propose several countermeasures to defend against this class of attacks.

CCS Concepts: • **Security and privacy** → **Side-channel analysis and countermeasures**;

Additional Key Words and Phrases: GPU, RSA, timing attack

ACM Reference format:

Chao Luo, Yunsi Fei, and David Kaeli. 2019. Side-channel Timing Attack of RSA on a GPU. *ACM Trans. Archit. Code Optim.* 16, 3, Article 32 (August 2019), 18 pages.

<https://doi.org/10.1145/3341729>

1 INTRODUCTION

The RSA algorithm [27] is a public-key cipher widely used in digital signatures and Internet protocols, including the Security Socket Layer (SSL) and Transport Layer Security (TLS). RSA entails

This work was done while Chao Luo was a PhD. candidate at Northeastern University.

This work was supported in part by the National Science Foundation under grants STARSS-1618379 and TWC-1563697, and by the Semiconductor Research Corporation under grant #2687.

Authors' addresses: C. Luo, MathWorks, Natick, MA, 01760; email: chaoluo@mathworks.com; Y. Fei and D. Kaeli, Northeastern University, Boston, MA, 02115; emails: yfei@ece.neu.edu, kaeli@ece.neu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/08-ART32

<https://doi.org/10.1145/3341729>

excessive computational complexity compared with symmetric ciphers. For scenarios where an Internet domain is handling a large number of SSL connections and generating digital signatures for a large number of files, the amount of RSA computation becomes a major performance bottleneck. With the advent of general-purpose GPUs, the performance of RSA has been improved significantly by exploiting parallel computing on a GPU [11, 22, 29, 32], leveraging the Single Instruction Multiple Thread (SIMT) model.

However, the security of RSA on a GPU has not received adequate attention. Side-channel vulnerability (i.e., private key retrieval through side-channel analysis) is a major concern on cryptosystems. Meanwhile, various side channels, including power, electromagnetic emanation, and timing, have been exploited for implementations of RSA on other computing platforms, such as CPUs, FPGAs, ASICs, and MCUs. Among these attacks, timing attacks have become a realistic cyber threat given that they are non-invasive and can be launched remotely. Many timing attacks of RSA deployed on CPUs are micro-architectural (cache-based) attacks [10, 25, 33, 34]. For example, in access-driven attacks such as PRIME+PROBE [25] and FLUSH+RELOAD [33], a spy process first sets the cache into a known state, the victim RSA process executes, and during the run the spy measures the timing of its memory accesses to infer the cache access pattern of the victim and recovers the secret key. We anticipate that applying such cache attacks on a GPU could be very challenging, if not impossible, due to the high degree of noise resulting from the massive parallel execution model of a GPU. These devices launch thousands of concurrent threads, so the victim would not be a single process, but a kernel that runs across threads on Streaming Multiprocessors (SMXes). The cache access pattern would be a complex mix of many copies of the same kernel, and therefore the spy cannot easily attribute a particular cache access to a specific victim thread. Each SMX has its own sets of caches, so the only cache leakage possible is when the spy kernel and victim kernel are on the same SMX. However, the victim kernels may be thread-switched out of the SMX by the hardware scheduler, which attempts to maintain high utilization. This results in a significant amount of noise in the cache access pattern that the spy kernel observes.

An alternative timing attack is time-driven—exploiting key-dependent variations in *computation* time and treating variations in memory accesses as key-independent noise. Such timing attacks heavily depend on the RSA algorithm and associated optimizations. There are two common optimizations used in RSA algorithms: a sliding window exponentiation and the Chinese Remainder Theorem (CRT). A Montgomery multiplication [21] is the basic operation that is common in many RSA implementations. Kocher [19] first outlined the theoretical foundation for possible timing attacks in several cryptosystems. Dhem et al. [7] proposed a practical timing attack of RSA on a smart card in a bit-by-bit fashion. Their attack exploits the variability present when using a Montgomery multiplication as the timing leakage, which is also adopted in this work. However, massive concurrent RSA executions on a GPU make this particular timing side channel much more obscure, presenting a challenge for our attack. Our GPU RSA implementation also employs sliding-window exponentiation, which was not used in prior work [7], making the attack based on recovering the exponent, window-by-window. Tóth proposed another timing attack under the assumption that the total number of extra reductions is known to the adversary [30], which is not a practical attack scenario on a GPU. RSA encryptions using the Chinese Remainder Theorem (CRT), and protocols based on CRT-RSA, have also been attacked using timing leakage [1–5, 28]. Our preliminary analysis shows that there may be no difference in the attack strategy for CRT-RSA on a GPU versus a CPU. Therefore, this article targets an RSA implementation that uses sliding-window exponentiation.

In this article, we extend our previous work [20] including two contributions. First, we construct a new attack method with sliding window of variable length nonzero window optimization. Previously, we only targeted constant length nonzero window optimization, which is a simpler case

to attack. Second, we build a success rate model to predict the effectiveness of the attack relative to the quality of the side-channel information.

The main contributions of this work include: (1) a hierarchical timing model for an RSA implementation on a GPU with Montgomery multiplication and sliding-window exponentiation, explicitly capturing various complex interactions in a massively parallel computing platform; (2) a successful correlation timing analysis attack based on the proposed timing model to extract the private key; (3) an effective error correction algorithm designed to detect and correct attack errors to improve the success rate; (4) a success rate analysis for success rate prediction with the quality of side-channel information obtained from the GPU.

The rest of the article is organized as follows: First, we discuss related work on GPU security in Section 2. In Section 3, we briefly introduce the RSA cipher, the optimizations, and a parallel implementation of RSA developed in CUDA and run on an NVIDIA GPU [11, 12]. In Section 4, we construct the timing model and propose a timing attack based on the model. We introduce the success rate analysis in Section 5 and present both empirical and theoretical success rates of attacks with different key sizes. In Section 6, we address the issues of attack error detection and correction. The experimental results are presented in Section 7. In Section 8, we propose effective countermeasures to protect RSA on GPUs. Conclusions are drawn in Section 9.

2 RELATED WORK

Naghibijouybari et al. [23] demonstrated the feasibility of constructing covert channels on a GPU by creating contentions on shared resources such as caches, functional units, and memory. The co-running trojan and spy can communicate covertly on the covert channel, forecasting the possibility of a side-channel attack. Because a covert channel relies on shared hardware resources (as well as co-running trojan and spy processes), such a side-channel attack would be classified as an access-driven type attack. In our timing-driven attack, we do not need to introduce additional spy load on GPU hardware resources, which makes our attack easier to launch and harder to detect.

Later work by Naghibijouybari et al. [24] builds on their earlier results [23], launching side-channel attacks on a GPU. Sensitive information, including user input to an OpenGL graphic application and internal parameters from a neural network application, was extracted from the GPU. In contrast to our work, this prior work requires a co-running spy process, where we do not. We also have a very different objective, recovering the secret key from a cryptographic algorithm.

In prior work by Jiang et al. [13, 14], two GPU-based timing-driven side-channel attacks are described. Our proposed work is distinctively different from them in two ways: First, in terms of timing leakage, our work exploits variations in computation time, while prior work exploited timing differences related to memory access, one due to the on-chip coalescing unit [13], and the other due to bank conflicts on an on-chip shared memory [14]. Second, our work targets a different cipher, RSA, while the prior work targets AES implementations. For AES (a block cipher), the data memory address pattern leaks key information. RSA is computationally intensive, and we anticipate such memory-access-related timing channels are very weak. Instead, we utilize a computational timing side channel for RSA on a GPU.

3 BACKGROUND: RSA AND GPU IMPLEMENTATION

An RSA cipher uses a pair of keys, one public and one private. The public key (n, e) is used for encrypting the plaintext message M , and the private key (n, d) is used for decrypting the ciphertext C :

$$M = C^d \mod n, \quad (1)$$

where n is the modulus, and e and d are the public and private exponents, respectively. The relationship between n , d , and e is that they have to satisfy constraints so the encryption and decryption are reverse operations to each other [27]. The key length (the bit length of C , M , d , and n) can be 512, 1,024, 2,048, or longer, providing different levels of security. Usually the public key exponent e is chosen to be a relatively small number with a small Hamming weight, while the secret key exponent d is much larger and requires many more bits. Therefore, the decryption process using the private key is 20 to 60 times slower than the encryption process, demanding acceleration for higher throughput [11].

3.1 Sliding-window Exponentiation

To perform exponentiation of a large number, a simple binary method [18] performs squaring and conditional multiplications based on each bit of the exponent. One optimization method is to perform *sliding-window exponentiation*, where the exponent d is decomposed into a series of zero and nonzero windows, F_i of length $L(F_i)$, and the exponentiation is processed window-by-window. Using sliding-window exponentiation, we can achieve a 21% improvement in performance on average [17].

There are two ways of constructing a sliding window: applying a Constant Length Nonzero Window (CLNW) [16] or a Variable Length Nonzero Window (VLNW), without incurring significant performance overhead [26]. For CLNW, the nonzero window length is fixed, while the zero window can have different lengths. For VLNW, both nonzero windows and zero windows have variable lengths.

Assuming Jang et al.'s implementation [11], CLNW has nonzero window lengths of 5, 6, and 7 for RSA-512, RSA-1024, and RSA-2048, respectively. The following example illustrates the partitioning of an exponent, with $L = 4$, from right to left:

1001 1101 00000 1011 0111 000 0001 0 0111 000 1011

Note that the exponent starts from most-significant non-zero window, and the value of the nonzero windows must be odd, while the length of the zero windows vary, and any zero window occurs between two constant-length nonzero windows.

For VLNW, the partitioning between zero and nonzero windows is slightly more complex. The nonzero window length has an upper bound, L_{max} . It switches to a zero window when the next q bits are all zeros. The algorithm proceeds from the least significant bit to the most significant bit, followed by the status of **ZW**, **NW**, indicating a zero window and a nonzero window, respectively.

ZW: Check the next bit. If it is zero, stay in ZW; else go to NW.

NW: Next, check $\min(q, L_{max} - L_{cur})$ bits. L_{cur} is the current window length. If all are zero, switch to ZW. If not, include the bits into the current window. If $L_{cur} = L_{max}$ after including the new bits, check the incoming single bit: if zero, go to ZW; else go to a new NW.

The following example illustrates the partitioning of an exponent, with $L_{max} = 5$, $q = 2$:

101 11101 00 101 10111 000000 1 00 111 000 1011

Note that any zero window occurs between two nonzero windows, and there may be multiple nonzero windows in a sequence. The nonzero windows will only be at odd value (not even). In Jang et al.'s implementation [11], $L_{max} = 4, 5, 6$, and $q = 3, 2, 2$, respectively, for RSA-512, RSA-1024, and RSA-2048. The left-most window will be zero-padded if its length is smaller than L .

The sliding-window exponentiation method for decryption is described in Algorithm 1, regardless of how the windows are partitioned. There are h windows of exponent d . To begin processing, the odd powers C^w of the method have to be pre-calculated once (Lines 1–4). The computation loop iterates over the most significant window. During each window (iteration), there are a number of squares computed (Line 7), which is equal to the bit length of the window, followed by conditional

multiplication (only if the window is non-zero) with the multiplicand determined by the window value, F_i (Line 9).

ALGORITHM 1: Sliding-window Exponentiation

Input: C, d, n (d decomposed into zero and nonzero windows F_i of length $L(F_i)$, for $i = 0, 1, 2, \dots, h-1$)
Output: $M = C^d \mod n$

```

1:  $C^2 = C \cdot C$ 
2: for  $w = 3, 5, 7, \dots, 2^L - 1$  do
3:    $C^w = C^{w-2} \cdot C^2 \mod n$ 
4: end for
5:  $M \leftarrow 1$ 
6: for  $i = 0$  to  $h-1$  do
7:    $M \leftarrow M^{2^{L(F_i)}} \quad // \text{squaring}$ 
8:   if  $F_i \neq 0$  then
9:      $M \leftarrow M \cdot C^{F_i} \mod n \quad // \text{multiplication}$ 
10:  end if
11: end for
12: return  $M$ 
```

3.2 Montgomery Multiplication

Montgomery's algorithm [21] is an efficient algorithm for modular multiplication, the basic operation used in the RSA cipher. The multiplier and multiplicand are first transformed into the Montgomery form before they go through the multiplication. The product is in Montgomery form as well: $\overline{a \cdot b} = \text{MM}(\overline{a}, \overline{b}) = \overline{a} \cdot \overline{b} \cdot r^{-1} \mod n$, where r is a power of 2, and co-prime with both n and $n < r$.

For decryption, the ciphertext comes in a Montgomery form. The decryption goes through a series of Montgomery multiplications using the sliding-window algorithm discussed in Section 3.1. Inside a Montgomery multiplication, there is a conditional reduction operation dependent on the values of $\overline{a}$ and $\overline{b}$. Because the extra reduction will take more execution time, this data dependency creates a timing side-channel attack surface. From prior work [28], if $\overline{a}$ and $\overline{b}$ are equal and random, which makes it a squaring operation, the probability of the extra reduction is:

$$\text{Prob}(\text{extra reduction in MM}(\overline{a}, \overline{a})) = n/3r, \quad (2)$$

which is a constant, and determined by n and r only. If $\overline{a}$ is random, and $\overline{b}$ is a constant, the probability becomes:

$$\text{Prob}(\text{extra reduction in MM}(\overline{a}, \overline{b})) = \overline{b}/2r, \quad (3)$$

i.e., dependent on values of $\overline{b}$ and r .

3.3 GPU Parallelization of RSA

NVIDIA GPU hardware features many Streaming Multiprocessors (SMXs), with each consisting of homogeneous cores and shared resources such as caches. In GPU programming, the workload is partitioned into a grid of blocks, with each block consisting of many threads. Threads from one block will only be executed on the same SMX. On CUDA cores, threads of one block are grouped into warps of 32 threads, executing instructions in a synchronized fashion. Whenever the execution path of threads in a warp diverges, only one path is executed at a time and the others have to wait. Control divergence negatively impacts program performance (i.e., the program

runs slower), resulting in data-dependent time variations. The execution time is dependent on the message contents and the private key. Warps on the same SMX are competing for the same limited hardware resources, such as CUDA cores, warp schedulers, and shared memory. For warps on different SMXes, they run independently.

We adopt a parallel RSA implementation developed in CUDA, as described by Jang et al. [11, 12]. Note that the *for* loop iterations in Algorithm 1 (Lines 6–11), when used for processing the exponent windows, are executed sequentially. This is because each iteration depends on the result of the previous iteration. Any parallelization is accomplished between messages and within messages. When processing a batch of ciphertext messages on a GPU, the messages are independent and decrypted in parallel while using the same secret key exponent. In addition, the Montgomery multiplications associated with one message's computations are also parallelized. Each message is represented by an array of s 64-bit words, and the multiplications are carried out by s threads, executed in cooperation. For example, to decrypt a single message in RSA-512, the s value is 8, i.e., it requires 8 threads to decrypt. A batch of 32 messages requires 256 threads in total, which are divided into 2 blocks, with a typical block size of 128 threads.

4 THE TIMING MODELS OF RSA ON GPUS AND A TIMING ATTACK

Given the parallel execution environment on a GPU, it becomes more challenging to predict the execution time versus a single-threaded execution on a CPU. Program execution includes both concurrent computation and blocking execution (e.g., for divergence in a warp). The execution time is not simply equal to the addition of all instructions anymore.

In this section, we introduce a hierarchical timing model for RSA decryption on a GPU, which accounts for various interactions during execution and is the basis for enabling our side-channel timing attack. Note that we choose the extra reductions in the Montgomery multiplication as the timing channel. The number of reductions depends on both the input message and each window of the secret exponent d . We start with a general linear timing model as follows:

$$T = O + v \cdot R, \quad (4)$$

where R is the number of extra reductions in the Montgomery multiplications, v is the unit execution time of one reduction, and O represents other key-independent execution time factors that follow a distribution when the input message varies.

For a sequential implementation, R will simply be the total number of extra reductions performed in each window. With multiple messages being decrypted in parallel by many threads on the GPU, there are complex interactions between the threads through competition for shared resources and the specific GPU programming model. We build a hierarchical model of R according to the interaction at different levels of parallelism on a GPU. We consider fine-grained thread (message) level interaction as well as interactions at a warp level, an SM level, and finally across the entire GPU.

4.1 GPU Timing Model

We begin by considering only a single message being decrypted by multiple threads (e.g., 16 for RSA-1024, which are within the same warp). Since threads in a warp are synchronized, we can still view the decryption for one message as sequential execution. For each window of d , we calculate the reduction number, as shown in Lines 7–10 in Algorithm 1. The first *for* loop (Lines 2–4) is ignored here, because it is independent of the secret key d .

For the i th window F_i of d , the intermediate variable M is first squared $L(F_i)$ times. If F_i is nonzero, its length $L(F_i)$ is always L when using the CLNW method. When F_i is zero, its length can vary from 1 to the key length. Under the assumption that each bit of d is independent and

uniformly distributed between 0 and 1, the longer the length of F_i , the smaller the probability will be. We assume the length of the nonzero window is no longer than 32 bits (probability of 2.33×10^{-10}). We use $r_{msg}(i)$ to record the number of reductions of the squares for window i . The number of reduction numbers performed for the entire decryption of message, R_{msg} , is the sum of $r_{msg}(i)$ over i .

Note, we do not include the extra reductions of the multiplication operation (Line 9 of Algorithm 1). Given that the probability of executing a reduction depends on the value of C^{F_i} , as expressed in Equation (3), there may be other windows with the same value of F_i . This kind of dependency on an extra reduction in multiple windows will bias the attack result [7], and an effective attack should not use the reductions associated with the multiplication.

Next, we consider running a warp using multiple messages in parallel. The number of messages in a warp depends on the key size. There are 4 messages in a warp for RSA-512, 2 for RSA-1024, and 1 for RSA-2048. Because threads in one warp are synchronized, the operations across different messages are also synchronized, (i.e., all the messages are processing the same window of size d at the same time). For one squaring operation in a window, if one or more messages have an extra reduction, the whole warp has to wait for it to finish, and the warp is considered to have an extra reduction. Thus, the number of reductions in a warp for a window $r_{warp}(i)$ is not simply the sum of the reductions used for squaring for all the messages.

To calculate $r_{warp}(i)$, we use a binary number, $b_{msg}(i, j)$, to record the extra reductions during execution, for the j th message in the i th exponent window. The length of $b_{msg}(i, j)$ is $L(F_i)$, because in each window there are $L(F_i)$ squaring operations, and each bit of $b_{msg}(i, j)$ is for the corresponding squaring. If the squaring of a message results in an extra reduction, this bit is 1; otherwise, it is 0. For each squaring operation, all the messages in a warp are synchronized (i.e., as long as there is one message resulting in an extra reduction, the warp (all the messages) encounters a delay). Therefore, all $b_{msg}(i, j)$ should have bit-wise OR operations over j , and the number of reductions in the warp, $r_{warp}(i)$ is the Hamming weight of the bit-wise OR result. The number of reductions across the warp for the entire decryption R_{warp} is also the sum of $r_{warp}(i)$ over i .

For one SMX with multiple warps running, there is no synchronization between warps. When the total number of threads (number of warps $\times$ 32) is equal to or smaller than the number of CUDA cores on the SMX, these warps will run mostly in parallel. However, the reduction operation involves shared memory accesses, which will be processed in a serialized manner, and therefore all the warps are processed in a blocking way to some extent. In scenarios where the total thread number is much larger than the number of SMX CUDA cores, the warps have to be scheduled serially. Based on these considerations, we let the reduction number of one SMX, R_{smx} , to be approximately equal to the sum of its warps' reductions.

For the complete decryption with multiple SMXes, we assume the SMXes are running independently, and the time of one decryption is determined by the slowest SMX. The number of reductions associated with one decryption on the GPU is R_{gpu} , which is the maximum of all the SMX reductions, R_{smx} .

4.2 Timing Model Verification

To verify our timing model, we run multiple experiments of RSA-1024 on an Nvidia K40 GPU. We set the block size to 128, such that each block will have 4 warps. We run experiments and vary the message number for one decryption from 1, 2, 8, to 64 with CLNW. We will have 1 message, 1 warp, 1 block (on one SMX), and 8 blocks (on 8 SMXes), respectively. For each experiment, we run the decryption 10K times with random messages, record the execution time of each decryption using the CPU's clock, and calculate the reduction number for the workload according to the timing model.

Fig. 1. Timing model verification.

Our results are shown in Figure 1, with the execution time normalized and correlation coefficients (ρ) of execution time and reduction number shown at the top. The figure shows we obtain a higher linear correlation between the measured execution time and calculated reduction number for (a) and (b) when we have one message and one warp running, respectively. The points in (c) and (d) spread out more for multiple warps and multiple SMXes due to the approximation of the reduction number for one SMX. We also characterize the timing channel for the VLNW implementation, and the correlation results are similar.

4.3 Correlation Timing Attack

In this section, we propose an attack method for the RSA GPU implementation based on the timing models we built in Section 4. We assume the ciphertext message is random but known to the attacker. For each decryption run, the adversary only records the total execution time.

The secret key d is extracted one window after another, starting from the most significant one, following the same order as decryption, as shown in Algorithm 1. For the i th target window, the previous attack results are used to calculate the value of the intermediate variable M at the beginning of i th iteration. To avoid confusion with M in other steps in the algorithm, we denote it as M_{temp} . For the first window, M_{temp} is 1.

In the attack, we make a guess of the target window. Based on the guess, we calculate the corresponding number of reductions. We compute the Pearson correlation coefficient between the number of reductions and the observed timing information. To determine the correct value for a nonzero window or length for a zero window, we iterate over all possible guesses for the target window and pick the one with the largest correlation based on our attack results.

4.3.1 Attack CLNW. We first describe the details of the attack of CLNW. To calculate the number of reductions for the i th window for one message, we assume there are at least two windows after the target window. Because the window could be zero or nonzero, we divide our guess into

Fig. 2. Operations on M_{temp} .

two groups, the length of zero windows and values of non-zero windows, which differs from previous work [7], which only works bit-by-bit. If the window is zero, its length, L_x , is varying, and we tally the number of L_x squarings i of M_{temp} first. For the sliding-window algorithm, a zero window is always followed by a nonzero window. Therefore, the next window must be nonzero of length L , resulting in another L squaring in the next iteration, but without any multiplication in between due to the zero window. From another perspective, the operations can also be considered as L squaring operations, followed by L_x squaring operations.

If the window is a nonzero window, we guess its value as V_x (V_x is an odd number). The following operations on M_{temp} are L squaring operations and one multiplication by C^{V_x} . Beyond the multiplication, we can still infer the operations by checking the next one or two windows. If the next window is nonzero, the operation beyond the multiplication is another L squarings, because the length of the next nonzero window is L . If the next window is zero, the window after the next window must be nonzero, and we have more than L squaring operations after the multiplication, consisting of the squarings of the next zero window and L squarings of the next nonzero window.

In conclusion, the first group of operations on M_{temp} are L squaring operations. After that, if the window is zero, the subsequent operations involve L_x squarings; if the window is nonzero with a value of V_x , the subsequent operations are multiplications with C^{V_x} and L squarings, as shown in Figure 2.

When calculating the number of reductions, we exclude the first L squaring operations, because they are common to all guesses. For a zero window, we record the extra reductions of the next L_x squarings in $r_{zero}(L_x, i, j)$ for each message, where i is the window index, and j is the message index. For a nonzero window, we record the extra reductions for the next L squaring operations in $r_{nonzero}(V_x, i, j)$, where the value V_x is used to compute the multiplications C^{V_x} .

We then subsequently calculate the number of reductions for the warps, SMXes, and entire GPU for the target window, using the timing model described in Section 4. Note that in our attack model, we are attacking window-by-window, and the calculated number of reductions is only for the current window. While in Section 4.2, the linear correlations are for the sum of the number of reductions across all windows. In our attack, we retrieve the private key window-by-window, where each window will have a lower correlation coefficient than the correlation across all h windows, as shown in Figure 1. In our experiments, the average Pearson correlation of attacking one window is about 0.02 when using 8 messages in one RSA-1024 decryption running on an NVIDIA Tesla K40 GPU, for 100K traces. The correlation value is much lower than that in Figure 1. However, it suffices to extract the correct key.

4.3.2 Attack VLNW. We extend our previous work [20] by adding the analysis of RSA with VLNW. Attacking VLNW is more challenging than attacking CLNW, because the length of the

Fig. 3. Operations on M_{temp} with VLNW.

nonzero window is no longer constant. To accommodate for this issue, we do not separate attacking zero and nonzero windows anymore. Instead, we combine the nonzero window with its preceding zero window as one attack *unit* as shown in Figure 3. If a nonzero window is preceded with another nonzero window, itself is an attack unit. In the attack, we first target the length of one attack unit, L_x , which equals to the length of the nonzero window, adding the length of the preceding zero window, if it exists. Then we target the value of the attack unit, V_x , which is also the value of the nonzero window. By knowing the length and value of each attack unit, we can reconstruct the private key.

Here we have another observation about the length of an attack unit: it is at least $q + 1$ bits long (q is defined in Section 3.1). The $(q + 1)$ -bit unit only happens when q zeros are followed by a single one bit. For other cases, the attack unit length is always longer than $q + 1$. During window partitioning (from right to left), if an NW switches to a ZW, there must have been q zeros, or the NW already has L_{max} bits. And we assume, $L_{max} > q + 1$.

To attack the i^{th} unit, we assume the previous $(i - 1)^{th}$ units are already known. The intermediate value of M_{temp} can be calculated at the beginning of the processing of the i^{th} unit. We first guess the length of the unit L_x by squaring M_{temp} L_x times and calculating the number of reductions for each message, the same as was done for CLNW. The value of L_x is determined by the correlation attack, similar to the zero window attack for CLNW. Next, we guess the value of the unit V_x , by multiplying the $M_{temp}^{L_x}$ by C^{V_x} and then squaring the result $q + 1$ times and calculating the number of reductions. Because the next attack unit has to be at least $q + 1$ bits, we assume the computation would be correct if we have the correct guess of V_x . By knowing L_x and V_x of the unit, we can recover the zero and nonzero windows and the private key.

The effort to launch a successful attack on VLNW is more than a successful attack on CLNW for two reasons: First, for each attack unit, we have to decide its length, which is prone to error, as we discuss in Section 6. Second, to decide the value of an attack unit, we calculate the reduction number from the $q + 1$ squarings, which is usually a small number. Therefore, the correlation when using the correct and wrong values is harder to differentiate.

5 SUCCESS RATE ANALYSIS

The ability to estimate the success rate of an attack would be useful when assessing the security of a system and making decisions on countermeasure implementation. In this section, based on the timing model and the correlation attack method presented in Sections 4 and 4.3, respectively, we build a probability model to predict the success rate of the attack. This success rate analysis extends

our previous work [20] that the number of traces needed for a successful attack can be estimated. Since the attack methods of CLNW and VLNW are similar, we only present an analysis for CLNW.

We target our model for the case of single-warp execution: there are m messages being encrypted in a single warp. Single message execution is just a special case of one warp execution with $m = 1$. For the multiple warps and the multiple SMX cases, the prediction will be less accurate, considering the low correlation shown in Figures 1(c) and (d). As a result, we only present the success rate analysis for single warp execution here, and we only analyze the attack of the CLNW shown in Section 4.3.1.

We rewrite the timing model of Equation (4)

$$T = C + I + v \cdot R, \quad (5)$$

where R is the number of reductions for the target window, v is the execution time of one reduction, C is a constant, and I is the noise, which follows a Gaussian distribution $\mathcal{N}(0, \sigma)$.

The attack methods for the nonzero and zero windows are different. We derive the success rates of both methods. To simplify our analysis, we assume we know whether the window under attack is zero or nonzero. This is a valid assumption, because when the proceeding window is zero, the current window under attack is nonzero.

We first consider attacking a nonzero window by guessing the value of V_x . Assuming the correct window value is $V_x = h$, the actual reduction number calculated from the timing model is $R = R_h$. For other incorrect values of V_x , we index h_i , with $0 \leq i < N_k - 1$, and N_k is the number of guesses for the window, which is 2^{L-1} , because V_x is an odd number. The number of reductions, corresponding to the wrong guesses, is R_i .

From the statistical success rate model proposed by Fei et al. [8], the success rate of attacking one window is:

$$SR = \Phi_{N_k-1} \left(\sqrt{N} \frac{v}{2\sigma} \mathbf{K}^{-1/2} \boldsymbol{\kappa} \right), \quad (6)$$

where Φ_{N_k-1} is the cumulative distribution function of a $N_k - 1$ dimensional standard Gaussian distribution; N is the number of traces; v and σ are the same parameters as in Equation (5). $\mathbf{K}$ is a three-way confusion matrix, whose element at index (i, j) is defined as:

$$\mathbf{K}_{i,j} = E[(R_h - R_i)(R_h - R_j)]. \quad (7)$$

$\boldsymbol{\kappa}$ is a confusion vector of size $N_k - 1$, whose i th element is defined as:

$$\kappa_i = E[(R_h - R_i)^2]. \quad (8)$$

To calculate $\mathbf{K}$ and $\boldsymbol{\kappa}$, we need to know the distribution of R . With a nonzero window length of L , there will be L squarings, each of which can possibly generate one extra reduction. Based on the analysis of the timing model for single-warp execution, for one square computation to generate an extra reduction, it requires that any of the m messages' square computations has an extra reduction. We assume the message values are randomized, which means the probability of an extra reduction is $n/3r$, as indicated by Equation (2). With $n \approx r$, the probability is close to $1/3$. If we assume messages are independent, then the probability of an extra reduction in the warp for one squaring is $1 - (1 - n/3r)^m$. There are L squares computed, so the distribution of R is a binomial distribution, with parameters $n = L$ and $p = 1 - (1 - n/3r)^m$:

$$R \sim B \left(L, 1 - \left(1 - \frac{n}{3r} \right)^m \right), \quad (9)$$

$$P(R = k) = \binom{L}{k} p^k (1 - p)^{L-k}. \quad (10)$$

Fig. 4. Theoretical and empirical success rate.

For different guesses of V_x , the intermediate results are random. We assume R_h , R_i , and R_j are independent. From probability theory,

$$E[(R_h - R_i)(R_h - R_j)] = 0, \quad i \neq j, \quad (11)$$

$$E[(R_h - R_i)^2] = \sum_{s=0}^k \sum_{t=0}^k P(R_h = s)P(R_i = t)(s - t)^2. \quad (12)$$

As a result, we have $\mathbf{K}$ as a diagonal matrix of $E[(R_h - R_i)^2] \times I$, and κ has $N_k - 1$ identical elements of $E[(R_h - R_i)^2]$.

From our experiments, we have found that $v/\sigma = 0.061, 0.043$, and 0.025 for RSA-512, RSA-1024, and RSA-2048, respectively. The success rates of attacking a nonzero window for RSA-512, RSA-1024, and RSA-2048 when running one warp are shown in Figure 4(a). Our experiment results show that, as the key length increases, the success rate decreases. The trend is because as the key size increases, there will be more windows. Our attack targets one window at a time, while using the total execution time. Therefore the signal-to-noise ratio drops with larger key size. The predicted success rates track the empirical success rates well, with a little over-estimation of the effectiveness of the attacks.

When attacking a zero window, the window length, L_x , is variable from 1 to the key length. The window length L_x is also a random number of geometric distributions, with parameter $p = 0.5$.

$$P(L_x = k) = 0.5^{-k} \quad (13)$$

Because the probability of the occurrence of a zero window that is longer than 32 is close to zero, we only make guesses between 1 and 32 during the attack. We assume the correct window length is L_h , with the value between 1 and 32, and the corresponding number of reductions is R_h . We index the other 31 guesses as L_i , $0 \leq i \leq 30$, and the corresponding number of reductions is R_i . There is a difference from attacking the nonzero window. When a wrong guess of V_x is made, the calculated number of reductions is independent of the correct one. However, with a wrong guess of the L_x , the calculated number of reductions is still partially correct. If $L_i < L_h$, all the computation of the reductions is correct but does not include the reductions beyond the L_i squarings. If $L_i > L_h$, the computation of the number of reductions is correct in the first L_h squarings. As a result, the distribution of $R_h - R_i$ is a binomial distribution, with the parameter values $n = L_h - L_i$ and

Fig. 5. Sequence of correlation coefficients of a timing attack when an error happens.

$p = 1 - (1 - n/3r)^m$. The success rate is dependent on L_h . The final success rate of attacking zero windows is a weighted average of the attacking different window lengths:

$$SR = \sum_{L_h=1}^{32} P(L_x = L_h) SR_{L_h}. \quad (14)$$

We plug in the distribution of $R_h - R_i$ into Equations (7) and (8), $\mathbf{K}_{i,j}$ and κ_i can be calculated for window length L_h , and the success rate SR_{L_h} using Equation (6). Note the value of $\mathbf{K}$ and κ are dependent on the correct window length L_h . The success rates for attacking zero windows are plotted in Figure 4(b). Results show that the success rate of attacking zero windows is much lower than nonzero windows with the same number of traces, indicating that our ability to distinguish zero windows is much lower than that of nonzero windows. In the attack of a nonzero window, the wrong guess of a window value will yield a totally incorrect number of reductions. For a zero window, if the window-length guess is close to the correct window length, the reduction number calculation is also very close. As a result, it is much harder to differentiate the correct window length from those close to it.

6 ERROR CORRECTION

The inherent drawback of the proposed attack, due to the iterative processing shown in Section 4.3, is that the attack on the target window is highly dependent on attack results on the previous windows. If any of the previous guesses were wrong, the current attack may not succeed because of the calculation M_{temp} and the reduction number will be wrong. As a result, the correlation coefficients after the first error will be much lower than those before, as can be observed in Figure 5. Each point in Figure 5 is the maximum correlation coefficient of one window attack when we iterate over all possible values. This non-robust feature can be utilized to detect errors if there exists N consecutive window attacks with low correlation coefficients.

In our attack, the length of a zero window is much harder to attack. Because the number of extra reductions is roughly linear with the bit length of the zero window, the ability to differentiate provided by the correlation attack is fairly small. For example, when guessing the length, the correlations of the wrong length guesses is very close to that of correct ones, as shown in Figure 6(a)

Fig. 6. Correlation coefficients of attacking zero and nonzero windows.

for the case of attacking a zero window, where the length of the window is 4. If a wrong guess ends up with the highest correlation, the error can only be detected in later windows. We define a parameter *distinguishability* D to indicate how likely an error can happen. The value of D for one window is defined as the normalized difference between the largest correlation and the second-largest correlation value. The lower D is, the more likely an error will occur. Based on these observations, we propose two error-correction methods: forward and backward methods.

Forward error correction is triggered when the distinguishability D of a target window is below a threshold D_{th} . We need to reconsider those values that were guessed and did not produce the highest correlation. We proceed with tentative attacks on the next window multiple times by setting the target window's value to other possible values, one after another. For each of those tentative attacks on the next window, we obtain a maximum correlation coefficient. To pick the right value for the target window, we choose the one resulting in the largest maximum correlation in the next window.

Backward error correction is triggered when there are N consecutive window attacks with the correlation coefficients below C_{th} . The attack rolls back to the first window of the N consecutive windows. This time, we choose the value for this window with the next highest correlation and then proceed. If this still results in an error and the window is attacked again, the next ranked guess should be chosen next. If all possible values for the window have been attempted and there is still an error, we move further backward to the previous window and reconsider other guesses.

Parameter selection: In our error-detection method, there are three parameters, D_{th} , N , and C_{th} . Their values affect the effectiveness and convergence speed of the attack. For D_{th} , if it is set too high, the forward error correction will be launched too often (false positive) and the attack will be impacted. If it is too low, the chance of overlooking an error is high (false negative). The error is only detected by the backward error correction, which is much slower than the forward one. So, a suitable D_{th} should balance the speed of the attack and the error-detection rate. Similarly both error-detection rate and attack speed will be affected by the N value. The choice of C_{th} has a similar impact as N , but in the opposite direction. This means a larger C_{th} is equal to choosing a smaller N . In our experiments, these values are chosen to maximize the success rate, with the attack speed in an acceptable range.

Table 1. Attack Result with Error Correction

Threads#	32			512			1024		
	FW	BW	E	FW	BW	E	FW	BW	E
RSA-512	0	0	1	4	2	3	3	2	1
RSA-1024	7	2	2	10	5	1	5	19	3
RSA-2048	12	1	0	7	16	0	24	13	0

FW/BW: forward/backward error corrections; E: errors left.

7 EXPERIMENTAL RESULTS

We implement the RSA cipher on an Nvidia K40 GPU, which has 15 SMXes and 192 CUDA cores on each SMX, hosted on a workstation running Ubuntu 14.04.05 LTS with an Intel E5-1603 processor. We record the timing information using the CPU clock that runs at 2.8GHz. We use the CPU instead of the more accurate GPU clock, because normally it is not accessible to the attacker, and we want to launch a realistic attack scenario.

We run experiments with different key lengths of 512, 1,024, and 2,048. The number of threads launched in one decryption varies from 32 to 1,024, varying GPU utilization from a single warp, a single block, to multiple SMXes. For each experiment, 100K decryption runs are carried out with random input messages. The error-correction parameters D_{th} , N , and C_{th} are set to 0.2, 8, and 0.01, respectively. The time used for one attack ranges from around 15mins for RAS-512 with 32 threads, to 8h for RSA-2048 with 1,024 threads. The larger the key size and the number of threads, the longer the attack time due to the increased computational complexity, and a larger number of error corrections needed.

Table 1 shows the number of errors corrected by forward and backward error correction, and how many windows (least significant/right-most) are incorrect after the attack. In the results, we see the number of errors increases as the number of threads and the key size increase. The forward error correction works best with fewer threads and a smaller key size. The backward error correction works best when using a larger number of threads and for a longer key. The errors left in our experiment are very small (0–3 windows, 18 bits at most), and the full key can be easily recovered using a lattice attack [6] or brute-force attack. For the largest error in our experiment, the brute-force attack space is 2^{18} . It takes about 15mins to attack RSA-512 with 32 threads, and about 8h to attack RSA-2048 with 1,024 threads, due to the increase in key size and error corrections.

8 COUNTERMEASURES

For countermeasures against GPU timing attacks, Kadam et al. [15] introduced randomized coalescing techniques to mitigate the GPU coalescing attack [13]. Due to the degree of indeterminism introduced, the correlation between the execution time and memory accesses is reduced or eliminated. However, the timing leakage we target in our attack does not relay on coalescing memory accesses, which renders these countermeasures non-applicable here. However, there have been several countermeasures presented that protect a CPU from a side-channel timing attack while running RSA. They attempt to eliminate data-dependent variations associated with the reductions [9, 31] or mask the messages and exponent with random numbers [19]. Such countermeasures are general and are also applicable to GPU implementations. However, they will introduce a significant performance degradation, which is a major concern for GPU acceleration. We evaluate an always-reduce Montgomery multiplication countermeasure on a GPU, which increases the execution time by 4.5%. We also compare the Pearson correlation coefficient between the number of reductions and the execution time for one message and show the result in Figure 7. The correlation

Fig. 7. An always-reduce countermeasure.

Fig. 8. Using a random-assignment countermeasure.

is significantly reduced from 0.79, as shown in Figure 1(a), to 0.06, rendering the time leakage very small and hard to exploit.

Special countermeasures targeting GPU implementations can also be devised. For example, we can randomize the assignment of messages to threads and blocks such that the execution time cannot be correctly predicted by the attacker. With a randomized assignment countermeasure, the correlation of one block reduces from 0.54, as shown in Figure 1(c), to -0.0073 , as shown in Figure 8. There is little computational overhead introduced here, except for a random permutation of a sequence. The permutation incurs randomized memory accesses, which impact the data spacial locality adversely. Security and performance can be balanced by varying the assignment granularity

for randomization. We can also add more noise to the timing measurements with dummy random messages, at the cost of performance and throughput. If we insert one dummy random message for every three messages, the correlation between the number of reductions and the execution time of one warp is also reduced from 0.66 to 0.12, which also introduces, at most, 25% overhead. This approach is suitable in scenarios where the GPU is lightly loaded, so additional dummy messages can be computed on the unused resources.

9 CONCLUSIONS

In this work, we evaluate the vulnerability of an RSA GPU implementation to side-channel timing analysis. We build timing models for a sliding-window RSA implementation using Montgomery multiplications. We consider a parallel implementation and evaluate it on an NVIDIA GPU. We characterize the execution behavior, capturing the execution time due to the complex interactions among threads. Our proposed attack methods are designed based on the timing model. We obtain attack results on an NVIDIA K40 hardware for different key sizes and work across different levels of parallelism. Our results show that an RSA implementation on a GPU is vulnerable to side-channel timing attacks, calling for both general countermeasures and GPU-specific countermeasures.

REFERENCES

- [1] Onur Aciicmez, Werner Schindler, and Çetin K. Koç. 2005 Nov. Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 139–146.
- [2] Cyril Arnaud and Pierre-Alain Fouque. 2013. Timing attack against protected RSA-CRT implementation used in PolarSSL. In *Proceedings of the Cryptographers' Track at the RSA Conference*. 18–33.
- [3] Billy Bob Brumley and Nicola Tuveri. 2011. Remote timing attacks are still practical. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS'11)*.
- [4] David Brumley and Dan Boneh. 2003. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium (SSYM'03)*, Vol. 12. USENIX Association, 1–1. Retrieved from: <http://dl.acm.org/citation.cfm?id=1251353.1251354>.
- [5] CaiSen Chen, Tao Wang, and Junjian Tian. 2013. Improving timing attack on RSA-CRT via error detection and correction strategy. *Inform. Sci.* 232 (2013), 464–474.
- [6] Don Coppersmith. 1997. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *J. Cryptol.* 10, 4 (1997), 233–260.
- [7] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. 1998. A practical implementation of the timing attack. In *Proceedings of the International Conference on Smart Card Research and Advanced Applications*. Springer, 167–182.
- [8] Yungsi Fei, A. Adam Ding, Jian Lao, and Liwei Zhang. 2015. A statistics-based success rate model for DPA and CPA. *J. Cryptog. Eng.* 5, 4 (2015), 227–243.
- [9] Gaël Hachez and Jean-Jacques Quisquater. 2000. Montgomery exponentiation with no final subtractions: Improved results. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 293–301.
- [10] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. *IACR Cryptology ePrint Archive* 2015 (2015), 898. Retrieved from <https://eprint.iacr.org/2015/898>.
- [11] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the Symposium on Networked Systems Design and Implementation*. USENIX Association, 1–14.
- [12] Keon Jang, Sangjin Han, Seungyeop Han, and KyoungSoo Park. 2015. libgpcrypto. Retrieved from: <https://github.com/lwakefield/libgpcrypto>.
- [13] Zhen Hang Jiang, Yungsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 394–405.
- [14] Zhen Hang Jiang, Yungsi Fei, and David Kaeli. 2017. A novel side-channel timing attack on GPUs. In *Proceedings of the Great Lakes Symposium on VLSI*. ACM, 167–172.

- [15] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. 2018. RCoal: Mitigating GPU timing attack via subwarp-based randomized coalescing techniques. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 156–167.
- [16] Donald Ervin Knuth. 1998. *The Art of Computer Programming, Seminumerical Algorithms*, Vol. 2, Addition Wesley. Reading, MA.
- [17] Cetin K. Koç. 1995. Analysis of sliding window techniques for exponentiation. *Comput. Math. Appl.* 30, 10 (1995), 17–24.
- [18] Çetin Kaya Koç. 1994. *High-speed RSA Implementation*. Technical Report. RSA Laboratories.
- [19] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference (CRYPTO'96)*.
- [20] Chao Luo, Yungsi Fei, and David Kaeli. 2018. GPU acceleration of RSA is vulnerable to side-channel timing attacks. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 113.
- [21] Peter L. Montgomery. 1985. Modular multiplication without trial division. *Math. Comput.* 44, 170 (1985), 519–521.
- [22] Andrew Moss, Daniel Page, and Nigel P. Smart. 2007. Toward acceleration of RSA using 3D graphics hardware. In *Proceedings of the Conference on Cryptography and Coding*.
- [23] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on GPGPUs. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture*. ACM, 354–366.
- [24] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2139–2153.
- [25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference*. Springer, 1–20.
- [26] Heejin Park, Kunsoo Park, and Yookun Cho. 1999. Analysis of the variable length nonzero window method for exponentiation. *Computers & Mathematics with Applications* 37, 7 (Apr. 1999), 21–29.
- [27] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 7.
- [28] Werner Schindler. 2000. A timing attack against RSA with the Chinese remainder theorem. In *Proceedings of the International Conference on Cryptographic Hardware & Embedded Systems*.
- [29] Robert Szerwinski and Tim Güneysu. 2008. Exploiting the power of GPUs for asymmetric cryptography. In *Proceedings of the International Conference on Cryptographic Hardware & Embedded Systems*.
- [30] R. Tóth, Z. Faigl, M. Szalay, and S. Imre. 2008. An advanced timing attack scheme on RSA. In *Proceedings of the International Telecommunications Network Strategy and Planning Symposium*, Vol. Supplement. 1–9.
- [31] Colin D. Walter. 1999. Montgomery exponentiation needs no final subtractions. *Electron. Lett.* 35, 21 (Oct. 1999), 1831–1832.
- [32] Yang Yang, Zhi Guan, Huiping Sun, and Zhong Chen. 2015. Accelerating RSA with fine-grained parallelism using GPU. In *Proceedings of the Information Security Practice and Experience*.
- [33] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the USENIX Security Conference*, Vol. 2014.
- [34] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Proceedings of the International Conference on Cryptographic Hardware & Embedded Systems*.

Received January 2019; revised May 2019; accepted June 2019