Exploiting Bank Conflict-based Side-channel Timing Leakage of GPUs

ZHEN HANG JIANG, Facebook, USA
YUNSI FEI and DAVID KAELI, Electrical & Computer Engineering Department, Northeastern University, USA

To prevent information leakage during program execution, modern software cryptographic implementations target constant-time function, where the number of instructions executed remains the same when program inputs change. However, the underlying microarchitecture behaves differently when processing different data inputs, impacting the execution time of the same instructions. These differences in execution time can covertly leak confidential information through a timing channel.

Given the recent reports of covert channels present on commercial microprocessors, a number of microarchitectural features on CPUs have been re-examined from a timing leakage perspective. Unfortunately, a similar microarchitectural evaluation of the potential attack surfaces on GPUs has not been adequately performed. Several prior work has considered a timing channel based on the behavior of a GPU's coalescing unit. In this article, we identify a second finer-grained microarchitectural timing channel, related to the banking structure of the GPU's Shared Memory. By considering the timing channel caused by Shared Memory bank conflicts, we have developed a differential timing attack that can compromise table-based cryptographic algorithms. We implement our timing attack on an Nvidia Kepler K40 GPU and successfully recover the complete 128-bit encryption key of an Advanced Encryption Standard (AES) GPU implementation using 900,000 timing samples. We also evaluate the scalability of our attack method by attacking an implementation of the AES encryption algorithm that fully occupies the compute resources of the GPU. We extend our timing analysis onto other Nvidia architectures: Maxwell, Pascal, Volta, and Turing GPUs. We also discuss countermeasures and experiment with a novel multi-key implementation, evaluating its resistance to our side-channel timing attack and its associated performance overhead.

CCS Concepts: • **Security and privacy → Hardware security implementation**; **Side-channel analysis and countermeasures**; • **Computer systems organization →** *Single instruction, multiple data*;

Additional Key Words and Phrases: Side-channel security, GPU security, microarchitectural attack

ACM Reference format:
Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2019. Exploiting Bank Conflict-based Side-channel Timing Leakage of GPUs. *ACM Trans. Archit. Code Optim.* 16, 4, Article 42 (November 2019), 24 pages.
https://doi.org/10.1145/3361870

This work was done while Zhen Hang Jiang was a PhD. candidate at Northeastern University.
This work was supported in part by National Science Foundation under grants STARSS-1618379, MRI-1337854, and SaTC-1563697, and the Semiconductor Research Corporation T3S program under GRC Task 2687.001.
Authors' addresses: Z. H. Jiang, Facebook, Seattle, WA, 98109, USA; email: zjiang1@fb.com; Y. Fei and D. Kaeli, Electrical & Computer Engineering Department, Northeastern University, Boston, MA, 02115, USA; emails: {yfei, kaeli}@ece.neu.edu.

1544-3566/2019/11-ART42
https://doi.org/10.1145/3361870

1 INTRODUCTION

GPU devices were originally designed to perform efficient three-dimensional (3D) graphics rendering, with thousands of parallel processing cores and a cognizant Single Instruction Multiple Thread (SIMT) execution model. Since the birth of programmable shader cores and high-level programming language frameworks [6, 32], GPU devices have dominated general-purpose parallel computing environments. Cloud-based services, such as encryption/decryption of large datasets, can effectively leverage the parallelism found on GPU devices to deliver high throughput for security engines.

As GPUs process sensitive data, the side-channel security of GPUs has begun to receive attention from the community. Work by Luo et al. [27] examines the information leakage of a GPU through the physical power side-channel for the first time. Work by Naghibhiijouybari et al. [29] shows how the unrestricted memory model assumed on a GPU can be used to leak website information that the victim has visited. Moreover, GPU architectures incorporate a number of unique microarchitectural features to support high data throughput, but some of which become a source of side-channel timing leakage. Work by Jiang [13] and Karimi [16] explore the side-channel timing vulnerability of a GPU's on-chip coalescing unit, a GPU-specific microarchitecture to consolidate concurrent spatially local Global Memory access requests into fewer requests. Another study by Naghibijouybari et al. [28] explores the timing-based covert-channel on GPUs through contention on caches, computational units, and memory operations. More recently, Luo at al. [26] explores GPU-based timing vulnerability due to data-dependent *computation* time, targeting a different cryptographic algorithm—RSA.

In our prior work [14], we identified a novel memory bank conflict-based timing channel in GPUs and developed an effective differential timing attack to retrieve the secret key. We demonstrated the attack on an AES implementation (based on the OpenSSL 0.9.7 library) on an Nvidia Kepler GPU.

The GPU on-chip Shared Memory is an important hardware unit for alleviating heavy traffic to the off-chip device memory. It is designed to store data that are shared and frequently accessed by many running threads. To support SIMT execution and deliver high memory throughput in modern GPUs, the Shared Memory is divided into multiple memory banks (versus a monolithic bank), allowing multiple concurrent paths to the Shared Memory. With multiple memory access requests for different memory banks being serviced in parallel, the Shared Memory bandwidth is significantly increased. However, when multiple memory requests compete for the same bank, they have to be serviced in a serial fashion, as each memory bank provides a single access port. We refer to such case of multiple accesses competing for a single shared memory bank port as a bank conflict. Additional requests that try to access data in the same memory bank will have their requests queued and delayed. This scenario will result in a detectable delay, as compared to multiple memory accesses that resolve to different banks with no bank conflicts.

Not only GPUs, modern high-performance CPUs (e.g., Intel's SandyBridge and ARM's Cortex-A) are also designed with multi-banked L1 and L2 caches. Yarom et al. [40] and Jiang et al. [12] investigate how sensitive information can be leaked when a cryptographic application runs on a CPU with multi-banked caches. A GPU generates a much more complex access pattern to Shared Memory banks, and our prior work [14] is the first one that identified the memory bank conflict-based timing channel and exploited it for a successful timing attack.

The major contributions of this article over our previous conference publication [14] include the following:

(1) We quantify the effectiveness of our attack methodology using the success rate as a metric.
(2) We extend our timing analysis onto other Nvidia GPU architectures: Maxwell, Pascal, Volta, and Turing. We explore how non-blocking execution can hide timing leakage in Shared Memory and be used to prevent our attack.
(3) We propose a multi-key protection mechanism and evaluate its effectiveness in mitigating side-channel leakage and performance overhead.

The article is organized as follows. In Section 2, we provide background on the Advanced Encryption Standard (AES) algorithm, as well as the GPU memory hierarchy and execution model. In Section 3, we describe the threat model for our attack. In Section 4, we explore timing variations due to Shared Memory bank conflicts, i.e., discovering the memory bank timing channel. In Section 5, we describe our differential timing attack targeting table-based cryptographic algorithms, and attack an AES encryption running on an Nvidia Kepler GPU. We also apply our attack in more realistic settings. In Section 6, we extend our timing analysis on other GPU architectures, and explore how its non-blocking execution mode can hide timing leakage in the Shared Memory. In Section 7, we discuss feasible countermeasures to prevent the attack, and focus on a multi-key implementation of AES encryption. Finally, we conclude the article in Section 8.

2 BACKGROUND

We begin by describing the AES implementation evaluated on the targeted GPU platform, focusing on the memory hierarchy and execution model of Nvidia Kepler GPUs, a commonly used and energy-efficient GPU microarchitecture [32]. While we focus on Kepler GPUs initially, we will also present results of timing attacks on a GPU from each NVIDIA architectural family since Kepler.

2.1 AES Encryption

In this article, we evaluate the timing leakage vulnerability of a table-based cryptographic algorithm on a GPU. We use the 128-bit ECB mode AES encryption as an example, and the same attack strategy can be applied to other table-based cryptographic algorithms such as Blowfish [36].

Performing AES encryption on a GPU can deliver an order of magnitude higher throughput than that on CPUs [30], since AES encryption can be easily parallelized and GPUs can exploit high degrees of execution parallelism. To demonstrate the generality of our attack, we port the implementation of AES from a standard and widely used library, the OpenSSL 0.9.7 library, into CUDA code. Note that the ported implementation of AES is similar to the versions evaluated for performance in many other studies [1, 30]. We discuss other implementations of AES that are immune to our attack but incur performance degradation in Section 7.

To port the implementation, we need to decide where to store T-tables in the GPU memory hierarchy and how to assign encryption jobs to GPU threads. In our prior work [13, 16] we stored T-tables in the Global Memory unit, but the implementation becomes vulnerable to coalescing attacks [13, 16]. Since T-tables are constant data and shared by all threads, they are a good candidate to store in the Shared Memory unit. Multiple studies on GPU implementations of AES have demonstrated the advantages of using the Shared Memory unit for storing T-tables [1, 3, 8, 21, 30, 31, 33], and our work adopts this implementation. To assign encryption tasks to GPU threads, we transform the AES encryption procedure into a single GPU kernel, where each GPU thread processes one block encryption, independently. Each block consists of 16 bytes.

The AES algorithm is composed of nine rounds of SubByte, ShiftRows, MixColumn, and AddRoundKey operations, followed by the last round with only three operations (omitting the MixColumn one). For faster processing, the first three operations are integrated into T-table lookups in the first nine rounds. In the last round, a special T-table ($T4$) is referenced, then followed by byte

Table 1. List of Tested Nvidia GPUs

Architecture	Kepler	Maxwell	Pascal	Turing	Volta
Device Model	TESLA K40c	GTX 950m	TITAN X	GTX 1660 Ti	Tesla V100 PCIE

masking. Each encryption round requires one 16-byte round key. The ten round keys are generated by the key scheduler using one 16-byte user-specified master key. Knowing any round key, an attacker can compute the original 16-byte master key. Our attack strategy targets the last-round key. A code snippet of the last round operations generating the first four bytes of ciphertext is shown in Listing 1.

```
1  O0 = (T4[(In0 >> 24) & 0xff] & 0xff000000)^
2       (T4[(In1 >> 16) & 0xff] & 0x00ff0000)^
3       (T4[(In2 >> 8 ) & 0xff] & 0x0000ff00)^
4       (T4[(In3      ) & 0xff] & 0x000000ff)^ k0;
```

Listing 1. AES Last Round Code Snippet.

Variable $\mathbf{O_0}$ is the first four bytes of the 16-byte ciphertext. Each variable, $\mathbf{In_0}$ to $\mathbf{In_3}$, contains 4 bytes of the input state for the last round. A selected byte of each variable is used to index into the T-table to obtain a four-byte output, of which only one byte contributes to the final ciphertext. $\mathbf{k_0}$ is the first four bytes of the last round key. From the original algorithm, the last-round can be simplified to use byte-wise operations, as shown below:

$$c_j = SBox[s_i] \oplus rk_j, \tag{1}$$

where the input byte position, i, for the SBox operation, differs from the output ciphertext byte position, j, due to the ShiftRow operation. Each byte of the last round input state, s_i, can be calculated once the corresponding cipher and key bytes are known by:

$$s_i = SBox^{-1}[c_j \oplus rk_j]. \tag{2}$$

2.2 Nvidia GPU Memory Hierarchy

In this article, we describe our attack on an Nvidia Kepler K40 GPU in detail, though we extend our analysis onto other Nvidia GPUs with different architectures, demonstrating the broad application of our approach. The GPU devices used in this article are listed in Table 1. These Nvidia GPUs have a similar memory hierarchy, except some of them have a dedicated Shared Memory unit, whereas the Nvidia Kepler GPU does not. We will describe the major differences in these memory architectures in Section 6 and discuss how the differences can impact the effectiveness of our attack. In this section, we will focus on the memory hierarchy using the Nvidia Kepler GPU as an example, shown in Figure 1.

On the Nvidia Kepler GPU, there is an off-chip DRAM memory (device memory) that is partitioned into global, texture, and constant memory regions. Data in those memories are shared among all threads running on all 15 Streaming Multiprocessors (SMXs). Each SMX (each with 192 single-precision floating point cores) also has L1, texture, and constant caches. Data in those caches are private to the threads running on the SMX. In addition, there is a Shared Memory for each SMX, and only the block of threads that allocated specific data in Shared Memory can access that data. Also, each GPU thread owns an exclusive set of 255 registers to store the current thread state.

On the NVIDIA Kepler GPU, the Shared Memory and the L1 cache reside in the same physical memory storage, with the total size at 64 KB. The individual size of the Shared Memory and L1

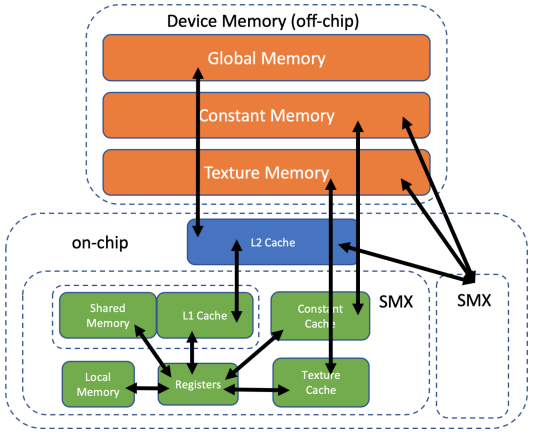


Fig. 1. Nvidia Kepler GPU memory hierarchy.

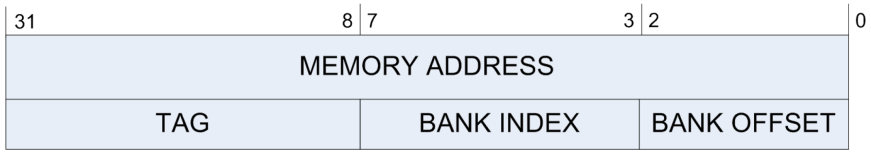


Fig. 2. Memory address to Shared Memory bank mapping.

cache is configurable. In our case, we allocate 48 KBs as the Shared Memory and 16 KB as the L1 cache. Note that the size configuration does not affect the attack, nor the results presented in this article.

The Shared Memory is divided into 32 banks, and it has a configurable bank line size (annotated as *banksize* in the Nvidia documentation [32]): four bytes or eight bytes. Since using a bank size of eight bytes will lead to fewer bank conflicts and improve an application's performance, we set the bank size to eight bytes for faster AES encryptions. The memory address breakdown for the Shared Memory is shown in Figure 2, where the three least-significant bits are used for the bank offset, and the next five bits (bits 3–7) are the bank index and are used for selecting the bank where a line is retrieved for kernel computation.

When multiple memory requests address different Shared Memory banks (i.e., bits 3–7 are different), they can be serviced in a single GPU cycle, providing much higher memory bandwidth than that of a monolithic cache bank design. However, a bank conflict occurs whenever multiple memory requests access the same bank (with the same bank index, but different tag values). Thus, there will be a noticeable timing difference between memory requests with and without bank conflicts.

2.3 Single Instruction Multiple Threads Execution Model

With the SIMT execution model, one GPU instruction is executed by at least a warp of 32 threads, and each thread has its own set of registers. However, all threads within a warp must be synchronized at an instruction boundary, which means that no thread in a warp can execute the next instruction until all threads complete the current instruction.

For memory instructions, each thread will generate a memory request. A warp of threads will generate 32 memory requests for one memory instruction. Under the SIMT model, the execution time of this memory instruction will be determined by the Shared Memory bank that receives the

highest number of bank conflicts (i.e., the largest number of requests resolving to the same bank). In other words, the execution time of a GPU memory access instruction becomes highly dependent on the memory addresses issued and whether these addresses result in bank conflicts. An attacker can exploit this dependency to recover the secret key of the cryptographic operations running on the GPU.

3 THREAT MODEL

Our threat model includes co-residence of the adversary and victim on one physical machine. We use this threat model for evaluation of our attack. However, this attack could just as easily be deployed in a cloud environment. The threat model assumes that the adversary is a regular user without root-level privileges, and the underlying operating system is not compromised. The adversary can measure the execution time of a GPU encryption kernel in a direct or indirect manner. For a direct measurement, the victim may expose the timestamp when a GPU kernel is launched and ended. For an indirect measurement, the victim can use non-privileged APIs to query the status of the GPU and infer the start and stop timestamps of the GPU kernel. A similar technique is described by Naghibijouybari et al. [29]. For the purposes of our evaluation, we will assume the victim exposes the timestamp whenever a GPU kernel is launched and when it finishes, providing direct measurements. The threat model also assumes that the adversary can observe the ciphertexts.

4 CACHE BANK CONFLICTS-BASED SIDE-CHANNEL TIMING CHANNEL

In this section, we conduct experiments to examine the impact of bank conflicts on GPU program execution time, i.e., discovering the *timing side-channel*. We develop a kernel that uses a warp of threads to issue loads to Shared Memory. Depending on the address of the data that each thread accesses, some number of bank conflicts will occur, resulting in different execution times for the load operations. We perform timing analysis on an Nvidia Kepler K40 GPU. All of the microbenchmarks presented are designed specifically for the microarchitecture of the Kepler memory system. Later, we show the same timing analysis on other architectures, such as Maxwell and Pascal, Volta, and Turing, which feature a range of memory hierarchies that differ from the Kepler architecture.

We develop a memory access pattern for a warp of threads to generate a specific number of bank conflicts, produced by selecting the address that each thread accesses. Using a high-resolution (a cycle-accurate timer) time-stamping mechanism, we can study the impact of bank conflicts on the kernel execution time. We have developed Microbenchmark 1, which is shown in Listing 2.

```
1  register uint32_t tmp, tmp2, offset = 64;
2  __shared__ uint32_t share_data[1024 * 4];
3  ...
4  int tid = blockDim.x * blockIdx.x + threadIdx.x;
5  tmp = clock();
6  tmp2 =share_data[tid * stride + 0 * offset];
7  tmp2 += share_data[tid * stride + 1 *offset];
8  ...
9  tmp2 += share_data[tid * stride + 39 * offset];
10 times[tid] =clock() - tmp;
11 in[tid] = tmp2;
```

Listing 2. Microbenchmark 1.

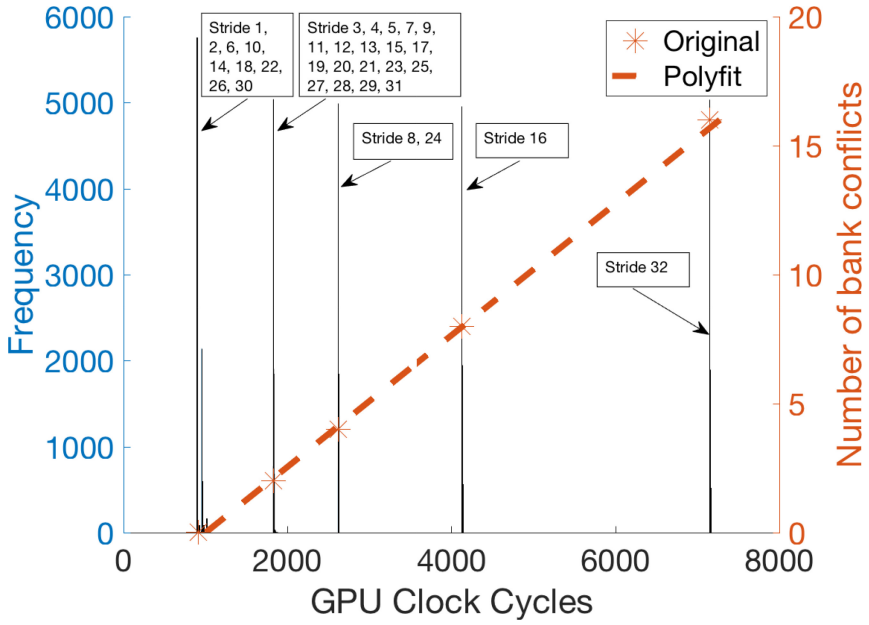


Fig. 3. The number of bank conflicts vs. the associated timing for 32 stride values.

The purpose of the microbenchmark is to run 32 concurrent threads in a warp, with each thread generating a sequence of memory accesses. We measure the execution time of the warp. In Listing 2, the variable *share_data* points to a continuous 16 KB of Shared Memory space, where each element of the array is one word (4 bytes). In Line 4, the thread ID is obtained. In Lines 6–9, each thread is accessing 40 memory locations in the sequence, with an offset of 64 words between two adjacent memory addresses (*offset*). Note the memory address distance between two threads is the *stride*, which can be fine-tuned to produce a different number of bank conflicts among a warp of threads.

Inspecting Listing 2 and Figure 2, we see that two adjacent memory addresses in a thread will have the same bank index and bank offset (64 words = 2^8 bytes), and therefore, all the memory addresses requested by a single thread access the same bank. Each thread accesses a single memory region, and the distance between memory regions accessed by the different threads is a single, or multiple *strides*. By selecting the value of the *stride*, we can create bank conflicts among threads in a warp. We run this kernel 320,000 times and collect 320,000 timing samples (10,000 timing samples for each stride value, ranging from 1 to 32). Based on our experiments, 10,000 timing samples are enough to produce a timing distribution. However, these timing distributions can shift depending on the system load during the experiments. The timing distribution for these timing samples is shown in Figure 3.

We observe that there are only five distinct timing distributions for the 32 stride values. Clearly, some stride values have the same timing behavior, and we suspect that those stride values result in the same number of bank conflicts. We next calculate the number of bank conflicts for each stride value. Recall that in our testing platform, the memory address breakdown is shown in Figure 2. Given a word index for the *shared_data* array, we can calculate the bank index by dropping the first least-significant bit, and then perform a modulo-32 operation, as described in the formula below:

$$idx_B = mod(idx_M >> 1, 32), \tag{3}$$

where idx_B is the bank index and idx_M is the array index. The right shift operator, >>, drops the least-significant bit, and the *mod* is the modulo operation.

As an example, assuming a stride value 16, for a memory access instruction issued across a warp of 32 threads, we will generate the following 32 memory indices: {0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256, 272, 288, 304, 320, 336, 352, 368, 384, 400, 416, 432, 448, 464, 480, 496}. Using Equation (3), we have the following bank access indices for the warp: {0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24, 0, 8, 16, 24}. Thus, these requests are for four banks, and each receives eight concurrent requests, i.e., eight bank conflicts produced when the stride is 16. Similarly, we calculate the number of bank conflicts for each stride value in the range of 1 to 32 words, and they end up in five groups, as shown in Figure 3. Each group corresponds to a different number of bank conflicts and associated average execution time.

We also plot the average execution time for a group (for selected stride values) versus the number of Shared Memory bank conflicts in Figure 3. We can easily identify a linear relationship. The slope of the linear line is 392, with an offset of 1,002 GPU cycles. Since we are performing 40 sequential Shared Memory loads, the result implies that the average penalty per bank conflict is 9.8 GPU cycles, which is also the strength of the timing channel signal in the Shared Memory banks. Although the penalty for GPU shared memory bank conflicts is not as large as the CPU cache miss penalty (another well-studied cache side-channel, with the difference between a hit and a miss around 100 cycles [10, 25, 39]), it can still be a source of information leakage. Countermeasures resistant to the original cache timing attacks may not work for the bank conflict timing channel. Next, we will demonstrate the feasibility of exploiting this fine-grained timing channel for key retrieval through statistical methods.

5 DIFFERENTIAL TIMING ATTACK

In this section, we devise a differential timing analysis attack to exploit the timing channel in Shared Memory banks. We start by attacking an AES algorithm, because its table lookup operations are key-dependent memory accesses. In our AES implementation, the lookup table is word aligned, similar to the *shared_data* array used in Microbenchmark 1. Therefore, we expect the execution time of one table lookup operation of a warp of threads to be linearly dependent on the number of bank conflicts generated by the threads. The execution time of one entire encryption is also dependent on the number of bank conflicts created by the table lookup operations.

Since the index for a table lookup operation is related to the round key, with the correct key guess, we can predict the number of bank conflicts that will occur during one round of AES encryption across a warp using Equation (3). By using many different blocks of plaintext, the correlation between the average encryption timing and the number of Shared Memory bank conflicts for the correct key guess should be high, and the correlation for incorrect key guesses should tend to be lower. This is the basic principle for a differential timing attack, similar to the traditional differential power attack (DPA) [19].

Next, we present the details of our attack methodology on AES. We specifically look at the mapping between the AES lookup tables and Shared Memory banks, as well as collect data and recover the last round AES encryption key.

5.1 Mapping between the AES Lookup Tables and GPU Shared Memory Banks

As described in Section 2, since we are attacking the last round of the AES encryption, we only need to examine the T_4 lookup table mapping in the Shared Memory. Note that attacking more rounds (more than three) becomes infeasible due to the algorithm-inherent statistical confusion and diffusion properties. There are 256 4-byte elements in the T_4 lookup table. Equation (3) is used to calculate the Shared Memory bank index, given a T-table lookup index, where idx_B is the bank

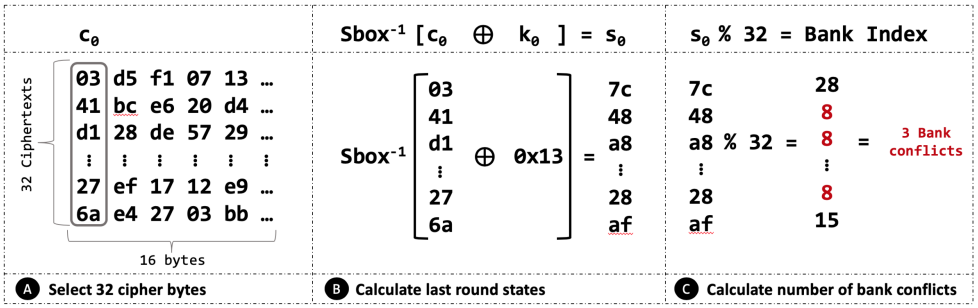


Fig. 4. Calculation from ciphertexts to number of bank conflicts.

index, and idx_M is the T-table lookup index. Because the bank size is 8-bytes in our Nvidia Kepler K40 GPU, we apply a right shift operator, >>, to drop the least-significant bit.

5.2 Collecting Data

The data collection procedure is similar to the experiments that we performed in Section 4. Instead of 40 memory load instructions, each thread performs an actual AES encryption using a random input data block. We record both the encryption time and the ciphertext for a warp of 32 threads. Each data sample composes of 32 16-byte ciphertexts and a timing value, as shown in the following format:

$$[\{\mathrm{C}^0, \mathrm{C}^1, \ldots, \mathrm{C}^{31}\}, t]$$
$$\textit{where each } \mathrm{C}^i \textit{ is a } 16-\textit{byte ciphertext produced by}$$
$$\textit{thread } i, \textit{ consisting of } 16 \textit{ bytes } \{c_0^i, c_1^i, \ldots, c_{15}^i\},$$
$$\textit{and } t \textit{ is the total encryption time for this warp}$$

We consider the encryption time measured from the GPU side and CPU side, respectively. The encryption time measured from the GPU side contains much fewer noise sources than that measured from the CPU side, because of the non-deterministic data transfer time between the GPU and CPU, as well as other required initialization procedures in the GPU device for running a kernel. However, the frequency of the CPU is much higher than that of GPU. Hence, the CPU-side encryption time provides more accurate measurements of the encryption. Moreover, measurements on the CPU side represent a more realistic scenario, as the adversary is just a passive observer and normally does not hold the GPU timer.

5.3 Calculating the Shared Memory Bank Index

For the last round of AES, with the output ciphertext known, the input state byte can be calculated using Equation (2). For a warp of 32 threads, 32 such table lookups are running concurrently, and therefore we have:

$$\left\{s_i^0, s_i^1, \ldots, s_i^{31}\right\} = SBox^{-1}\left[\left\{c_j^0, c_j^1, \ldots, c_j^{31}\right\} \oplus rk_j\right], \tag{4}$$

where c_j^0 is the cipher byte produced by thread 0, s_i^0 is the lookup table index for thread 0, and rk_j is the jth last round key byte, which is common to all the threads. These lookup table indices, $\{s_i^0, s_i^1, \ldots, s_i^{31}\}$, are exactly the Shared Memory indices. We use Equation (3) to further calculate the bank indices used by all threads in the warp for this table lookup instruction, and then derive the number of bank conflicts.

Using the example shown in Figure 4, assuming we have encrypted 32 16-byte plaintexts using 32 threads (i.e., a warp) and obtain 32 16-byte ciphertexts, and we are targeting to recover the 0th last round key byte. First (Ⓐ), we select the 0th cipher byte from the 32 ciphertexts. Second (Ⓑ), we convert the selected cipher bytes to the last rounds states using a guessed key byte value. Lastly (Ⓒ), we calculate the accessed Shared Memory bank indices and the number of bank conflicts that occurred. Note that if the key guess value is not correct, the number of bank conflicts calculated will be incorrect, and would not correlate with the observed timing.

5.4 Recovering Key Bytes

Using the collected data, we can launch a correlation timing attack. As shown in Listing 1, for the last round of AES each T-table lookup uses one byte in the 16-byte state, and therefore, each round key byte can be attacked independently. For each data sample we collected, we calculate the number of bank conflicts for the table lookup instruction that is using the jth last round key byte, as shown in the example in Figure 4. For each key byte value guessed (ranging from 0 to 255), we can calculate the correlation between the average timing and the number of bank conflicts, and use the correlation value to differentiate the correct key byte from other incorrect key guesses. For the data collected, the number of bank conflicts between the 32 threads falls in the range of [2, 4].

The power of a correlation timing attack lies in the linearity of the timing model, i.e., the total execution time should consist of a deterministic component, linearly dependent on the number of bank conflicts, and an independent Gaussian random variable contributed by the other nine rounds. During an actual AES execution, the timing distribution does not conform to the ideal model, and therefore a correlation timing attack may not be more effective than a differential timing attack, which only considers two values in terms of the number of bank conflicts.

Thus, we adopt a differential timing attack approach, and calculate the two average timing values, one for the group of data samples that generate two bank conflicts, and the other for the group of data samples that generate four bank conflicts. The Difference-of-Means (DoMs) between these two groups should be about two times the bank conflict penalty, i.e., around 19 cycles. Thus, for each sample we collected, we first calculate the number of bank conflicts as shown in Figure 4. Second, we classify its corresponding timing into one of two groups based on the number of bank conflicts. Finally, we compute the DoM between these two groups. If the correct key value were used, then we would see a DoM value of around 19 cycles. Otherwise, the DoM should be close to zero.

We first apply the attack method to the 15th key byte. The result is shown in Figure 5. The upper plot is using one-hundred thousand samples, and the lower plot is using 1 million samples. The correct key byte value (198) is highlighted in red. In the upper figure, the DoM for the true key value is 15.99 GPU cycles, which is about 4 GPU cycles less than the predicted signal, 19.6 GPU cycles, and the DoMs for other values are much smaller, between −2.3 and 2.3 GPU cycles. By increasing the sample size to 1 million, the DoM for the true value remains about the same, while the DoMs for wrong values are reduced to a range between −0.8 and 0.8 GPU cycles.

We apply this attack methodology and recover the other key bytes. The result is shown in Figure 6. All 16 true key byte values clearly stand out in the plots, which means we have successfully recovered all key bytes.

Although the same attack runs on all key bytes, some key bytes have much smaller peak timing differences, such as k_0 and k_1, as compared to other key bytes. We observe that the key bytes that are used closer to the end of encryption tend to have larger and distinct timing differences, e.g., k_{15}. We speculate that the reduced signal in k_0 is caused by instruction-level parallelism. Although the architectural details of the Nvidia Kelper GPU are not public, we suspect that the Nvidia Kepler GPU can continue issuing independent instruction(s) in each cycle until all resources are consumed, and therefore, it hides the latency due to Shared Memory bank conflicts by issuing and

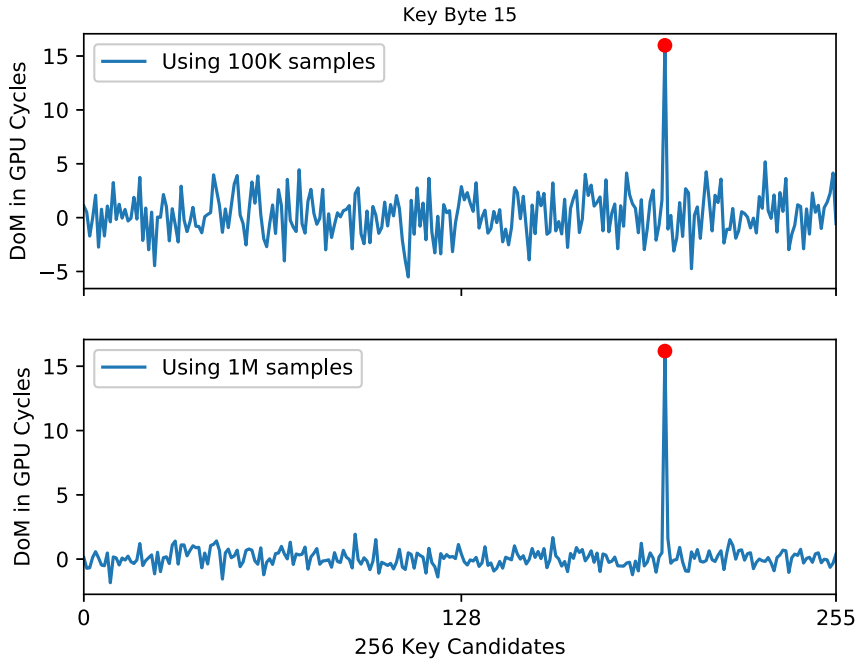


Fig. 5. The 15th key byte recovery using GPU timing information.

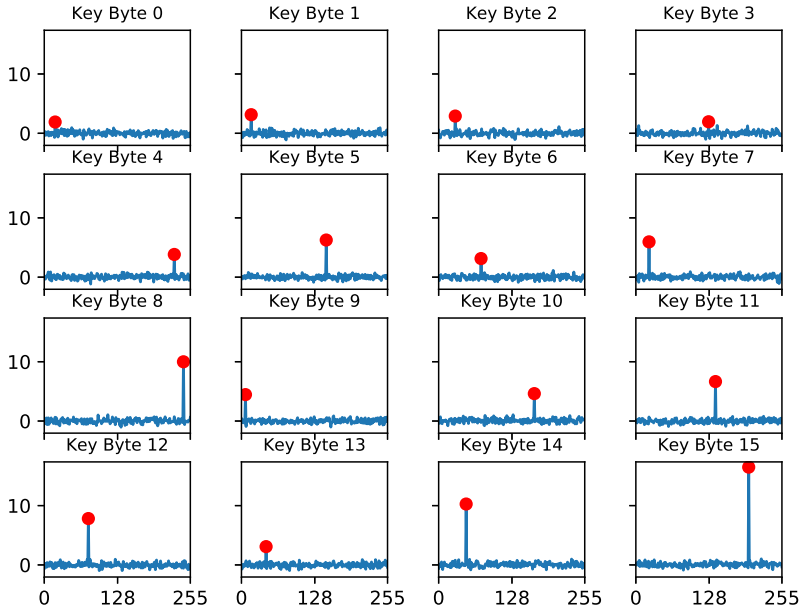


Fig. 6. Recovery of all 16 key bytes using 10 million samples.

executing other independent instructions. However, if the GPU stalls due to Shared Memory bank conflicts, the penalty is exposed in the total execution time, and therefore, we observe a stronger signal (e.g., key byte 15).

To verify our speculation, we slightly modify Microbenchmark 1, as shown in Listing 3, to remove the accumulation instruction after each load instructions, and therefore all the load instructions are independent on each other and can be scheduled in a non-blocking fashion.

```
1 ...
2 tmp2 =share_data[tid*stride+0*offset];
3 tmp3 = share_data[tid*stride+1*offset];
4 tmp4 = share_data[tid*stride+2*offset];
5 ...
6 tmp41 = share_data[tid*stride+39*offset];
7 times[tid] = clock() - tmp;
8 in[tid] = tmp2+tmp3+tmp4+...+tmp41;
```

Listing 3. Microbenchmark 2.

We run Microbenchmark 2 10,000 times for each stride value in the range of $[1, 32]$ and collect the timing information. We calculate the average timing for each stride value and the number of bank conflicts, and obtain a similar linear relationship, shown in Figure 3. However, the slope becomes 177 GPU cycles per 40 load instructions, and therefore the per-conflict penalty is reduced to 4.4 for Microbenchmark 2 from 9.8 GPU cycles for Microbenchmark 1.

In the modified kernel, each of 40 load instructions loads data into a different register, and at the very end, the addition is performed. The execution of Microbenchmark 2 is non-blocking, while the execution of Microbenchmark 1 is in a blocking mode due to tight data dependency. We show the SASS code (assembly code for the GPU kernel) for the two microbenchmarks in Listing 4 and Listing 5, respectively. In Microbenchmark 1, there is a strong data dependence between instructions, and the loaded data (Line 1 of Listing 4) is used by a later operation (Line 4) that is three instructions away from the load operation. Such read-after-write (RAW) data dependencies between Line 4 and Line 1 introduce blocking, and Instruction 4 cannot proceed until Instruction 1 is completed, exposing a possible delay that Instruction 1 may experience in the total execution time. This is seen in Figure 7(a), where the total execution time is the execution time of Instructions 1 and 4. In Microbenchmark 2, we generate a sequence of independent loads, as shown in Listing 5. Instructions can be executed in a non-blocking fashion. Therefore, the delay caused by Instruction 1 may be hidden by executing later instructions. We can see this in Figure 7(b), where the total execution time no longer depends on Instruction 1.

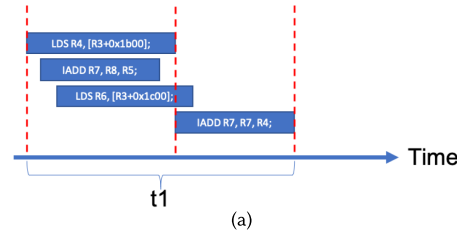
For our AES algorithm implementation, each lookup operation in the last round (16 lookups in total) is independent of one another. Since k_0 is being used in the first lookup operation, its delay due to bank conflicts is obscured by executing other independent lookup operations. Thus, we see a weaker signal for key byte k_0. While k_{15} is the last one to be processed, and the delay due to bank conflicts is completely exposed in the execution time, i.e., k_{15} has the strongest signal. Results in Figure 6 show that signals (the average penalty for one bank conflict) for all the key bytes range from 0.8 GPU cycles to 8.9 cycles, depending on how the conflict penalty is hidden by non-blocking execution mode.

5.5 More Realistic Attack Scenarios

We have demonstrated a successful attack using timing information taken from the GPU side, which contains much less noise than the timing information measured from the CPU side. Also, we evaluate our attack methodology when running AES encryption with 32 parallel threads. It

Listing 4. Original SASS Code for Microbenchmark 1

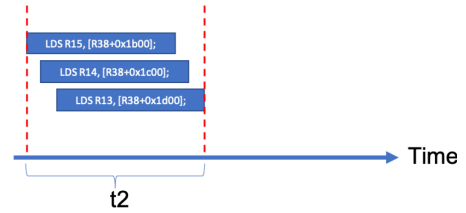
```
1 LDS R4, [R3+0x1b00];
2 IADD R7, R8, R5;
3 LDS R6, [R3+0x1c00];
4 IADD R7, R7, R4;
```



(a)

Listing 5. Modified SASS Code for Microbenchmark 2

```
1 LDS R15, [R38+0x1b00];
2 LDS R14, [R38+0x1c00];
3 LDS R13, [R38+0x1d00];
```



(b)

Fig. 7. Blocking mode vs. non-blocking mode.

is important to understand the scalability of the attack (i.e., how the number of traces needed for a successful attack grows as we increase the number of threads). In this section, we evaluate the effectiveness of our attack using CPU timing information, and evaluate the impact of noise in the measurement when encryption is run with a larger number of threads. These changes more accurately reflect a typical execution environment for a timing side-channel attack on a GPU.

5.5.1 Using CPU Timing Information. To collect timing data from the CPU side, we record the kernel execution time using an x86 instruction *rdtscp*. Since the CPU runs at a higher frequency than the GPU, CPU timings have higher resolution (4.8 times higher). However, the CPU timing information is much noisier than the GPU timing information. Using the CPU timestamping mechanism, we can record the time from when a GPU kernel is launched until it exits. Thus, CPU timing information inevitably includes extra timing noise due to the kernel launch overhead generated by memory transfers between the CPU and the GPU. Alternatively, using a GPU timestamping mechanism, we can record timestamps after the kernel launch is complete, only capturing when the AES encryption begins and ends inside the GPU kernel, avoiding extra noise introduced during the kernel launch.

We perform the same differential timing attack against our 32-thread implementation, and we can still recover all key bytes using 1 million samples, but with a weaker signal, as shown in Figure 8. Using the CPU timing information, the difference of means for the correct key value is around 94 CPU cycles (20 GPU cycles) between the two bins for the correct key guess.

We also quantify the effectiveness of our attack by introducing a metric, called *success rate*, which captures the probability of success of an attack given the number of traces. With the success rate metric, we can compare the effectiveness of the attack across different platforms and different side-channel attacks. There are other existing metrics [5, 7, 35], but they quantify the information leakage from the side-channel instead of the effectiveness of the attack. We perform a number of attacks to obtain the empirical success rates of recovering the 15th key byte when using the CPU timing information, and also when using the GPU timing information. The results are presented in Figure 9. Using the GPU timing data yields a stronger side-channel timing signal, and therefore only 120,000 samples are needed to achieve a 100% success rate, while 900,000 samples are needed when using the CPU timing.

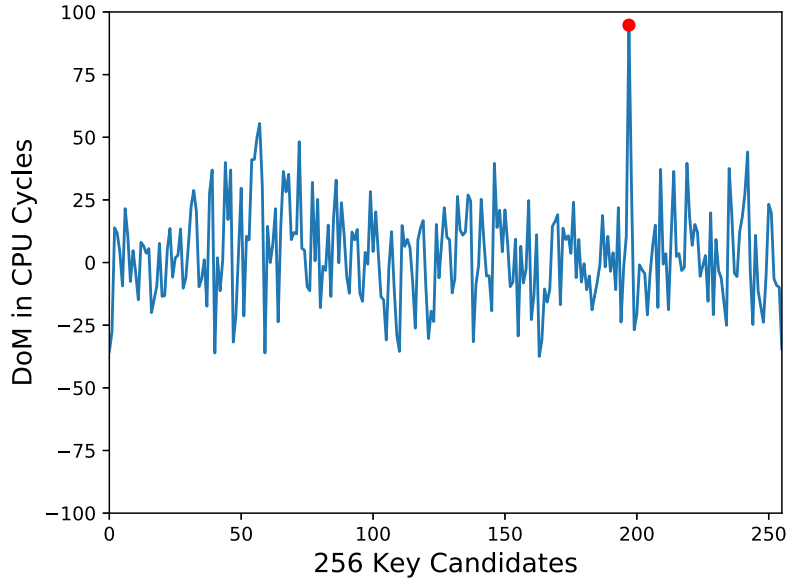


Fig. 8. The 15th key byte recovery using the CPU timing information.

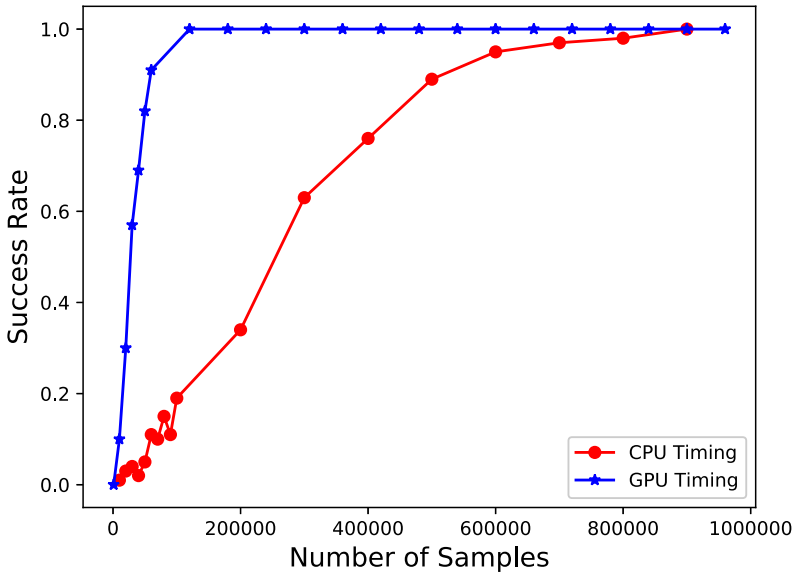


Fig. 9. The 15th success rate: CPU timing vs GPU timing.

5.5.2 Increasing the Number of Threads. Next, we evaluate the scalability of our attack by using an 8,192-thread AES implementation using 16 blocks of 512 threads (each block contains 16 warps), which is several times more than the maximum number (2,880) of threads that can run in parallel on the Nvidia Kepler K40 GPU. In realistic scenarios, we would want to keep the entire device busy with encryptions/decryptions, producing much higher throughput versus using only 32 threads.

Note that in such a real attack scenario, the attacker cannot manipulate the kernel code running on the GPU (they cannot easily insert timestamps before and after the encryption). Instead, they

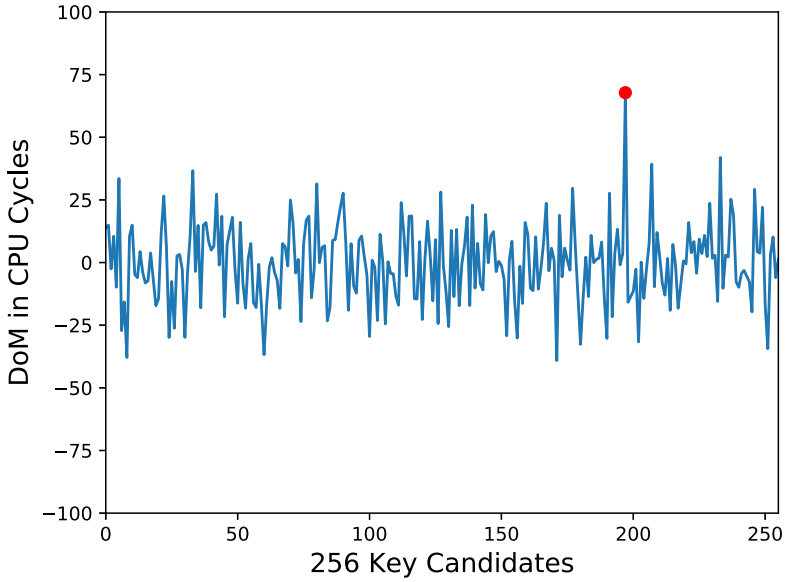


Fig. 10. The 15th key byte recovery using 1 million samples with full GPU capacity.

would have to rely on the timing information measured from the CPU side. When a higher number of threads are running, the measured execution time of the kernel is dominated by the slowest SMX (and the corresponding blocks running on it). However, the details of the GPU scheduler are not public, so we do not know exactly how blocks and warps are distributed and scheduled on the GPU SMXs.

We choose to select one warp to attack, and use the entire kernel execution time, which represents a real black-box passive attack scenario. We anticipate much higher noise when using this attack model, because the selected warp may not run on the slowest SMX, and even if it does, there are other warps competing for the same SMX resources. Attributing any variation in the kernel execution time to this warp is not accurate. This non-deterministic process adds a larger amount of noise into our data samples. Other activities, such as saving a thread's register state, can also contribute to the noise in timing.

We use the same attack methodology described earlier, collecting 1 million samples of the 8192-thread AES encryption, and we apply our differential timing attack to recover the 15th key byte. The result is shown in Figure 10. The timing difference for the 15th true key value is 67.8 CPU cycles, which barely stands out among the other wrong key values. This indicates there is a high degree of noise in the data samples we collect that result in a significantly lower signal-to-noise ratio (SNR).

Through both scenarios, we demonstrate that our attack can sustain the noise. Using our success rate metric, we show that by increasing the number of samples, our attack can overcome a higher degree of noise.

6 TIMING ANALYSIS ON OTHER ARCHITECTURES

So far in this article, we have focused our analysis on the Kepler architecture. We now extend our timing analysis onto other architectures: Maxwell, Pascal, Volta, and Turing. The memory hierarchies of both Maxwell and Pascal are quite different from that of the Kepler architecture. Similar to the memory hierarchy in the Kepler architecture, the Shared Memory and L1 cache

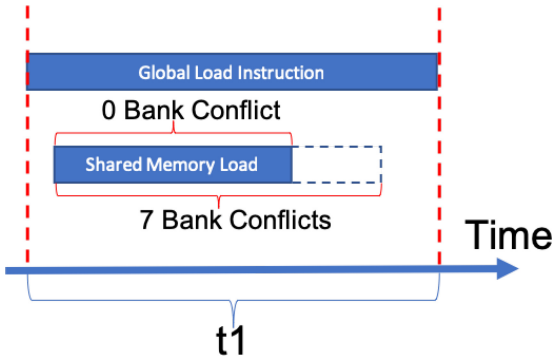


Fig. 11. Non-blocking mode in Maxwell and Pascal.

share the same physical unit on both Volta and Turing architectures. However, both Maxwell and Pascal architectures have a dedicated Shared Memory (separate from L1 cache). All of these newer architectures have the same number of Shared Memory banks as the Kepler architecture, but fix the bank size to 4 bytes.

Based on our initial timing analysis on these four architectures, the Shared Memory bank penalty is only 2 GPU cycles, which is significantly lower than the 9.8 GPU cycles found on the Kepler architecture. In theory, we should still be able to leverage the timing leakage as before. However, to our surprise, we could recover key bytes on both the Volta and Turning architectures, but failed to recover any key bytes on both the Maxwell and Pascal architectures. We summarize the attack results in Table 2. We speculate that the non-blocking execution mode completely hides the latency due to Shared Memory bank conflicts, as discussed in Section 5.4, where we observed a significant reduction in the signal-to-noise ratio.

Many of intermediate states (such as register spills) in the AES implementation are stored in Global Memory. On both Maxwell and Pascal architectures, the Shared Memory and L1 cache no longer share the same physical memory space, and hence Global Memory and Shared Memory loads/stores can be processed concurrently, which is not the case in the Kepler architecture. The latency for an off-chip Global Memory access (load/store) is longer than an on-chip Shared Memory access. By issuing a Global Memory access in parallel with a Shared Memory access, regardless of the number of bank conflicts the Shared Memory access induces, its penalty is completely hidden by the latency of the Global Memory access, as shown in Figure 11. However, on both Volta and Turing architectures, the Shared Memory and L1 are unified again, similar to that on the Kepler architecture, and Global Memory and Shared Memory loads/stores have to be processed in a sequential manner and the delay of shared memory bank conflicts are exposed in measurements. We were able to recover key bytes on both Volta and Turing architectures.

To test our reasoning, we design several benchmarks to test the effects of a Global Memory load (L1 cached) on the timing characteristics of a Shared Memory load. We first develop a kernel that generates a warp of threads, selecting the memory addresses so that the warp will induce a predetermined number of bank conflicts. The memory microarchitecture in all newer architectures has 32 Shared Memory banks, where each bank is 4-bytes wide, so accessing two memory addresses with the same address values for bits 2-6 will cause a Shared Memory bank conflict. We provide the C code in Listing 6.

The first loop in Listing 6 generates memory addresses for *num_conflicts + 1* threads in a warp and results in *num_conflicts* bank conflicts. The second loop generates memory addresses for the remaining threads in the warp and will not cause any bank conflicts. To determine the Shared

```
for (j = 0; j < num_conflicts + 1; j++)
  input[j] = j*128;
for (; j < warp_size; j++)
  input[j] = j * 4;
```

Listing 6. C Code for Generating Memory Addresses.

Memory bank conflict penalty, we simply record time for a Shared Memory load instruction using this set of memory addresses, as shown in Listing 7.

```
t0 = clock();
__threadfence_block();
shared_memory_array[input[tid]];
__threadfence_block();
t1 = clock();
```

Listing 7. Benchmark for Measuring the Bank Conflict Penalty.

Shared_memory_array is a 256-entry one-byte wide array that is allocated in the Shared Memory. *tid* is the thread ID, and each thread uses its ID to specify an index in the *input*. The *__threadfence_block()* function is a blocking instruction, which prevents threads to proceed without completing all prior instructions.

We first run these benchmarks on the Pascal architecture and collect 10,000 timing samples for each number of bank conflicts [0, 7]. We show the time distribution for each bank conflict value in Figure 12(a). We include a linear regression line for the number of bank conflicts versus the mean of its timing distribution. As we can see in Figure 12(a), the slope of the regression line is approximately 2 GPU cycles per conflict. We also calculate the correlation coefficient between the number of bank conflicts and the recorded time. The correlation coefficient is 0.95, which indicates a strong linear relationship between the number of bank conflicts and the execution time. However, if we insert a Global Memory load instruction before the Shared Memory load instruction (as shown in Listing 8), the linear relationship disappears (resulting in a 0.06 correlation value), as shown in Figure 12(b). Although we have added a Global Memory load instruction, the average time for each distribution only increases by 12 GPU cycles.

```
t0 = clock();
__threadfence_block();
global_memory_array[tid];
shared_memory_array[input[tid]];
__threadfence_block();
t1 = clock();
```

Listing 8. Benchmark for Measuring the Bank Conflict Penalty with a Global Memory Load Inserted Before a Shared Memory Load.

Furthermore, the effect is the same when the Global Memory load instruction is inserted after the Shared Memory load instruction, as shown in Figure 12(c). The execution time for the Global Memory load instruction dominates the total execution time, so regardless of the instruction order, as long as they are issued in parallel, the execution time of the Global Memory can completely hide the latency due to Shared Memory bank conflicts. By inserting a *__threadfence_block()* function call between the Global Memory and the Shared Memory load instructions, we can again reveal

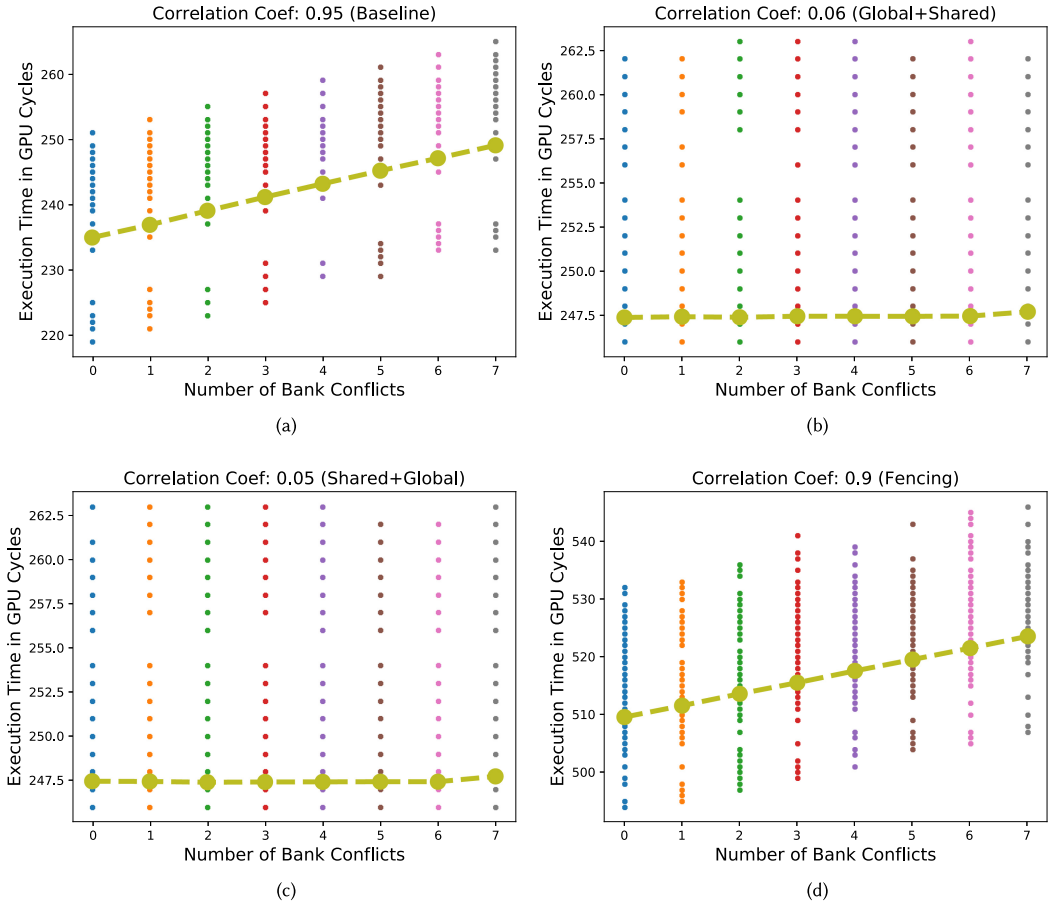


Fig. 12. Timing analysis of Shared Memory load instruction.

Table 2. Attacks on Different Nvidia Architectures: Number of Key Byte Recovered Using 1 Million Samples

Architecture	Unified L1+Shared Memory	# Key Byte Recovered	Penalty per Conflict (GPU Cycle)	G+S Load Corr	S+G Load Corr	G+F+S Load Corr
Kepler	Yes	16	9.8	0.95	0.95	0.94
Maxwell	No	0	2	0.00	−0.01	0.98
Pascal	No	0	2	0.06	0.05	0.99
Turing	Yes	16	2	0.99	0.99	1
Volta	Yes	10	2	1	1	1

G: global memory access; S: shared memory access; F: Fencing instruction.

the bank conflict penalty, as seen in Figure 12(d). With the fence function call, the total execution time is simply the sum of the execution time for each load instruction.

We run these benchmarks on all architectures, Kepler, Maxwell, Volta, and Turing, and summarize the benchmark results in Table 2. The results show that a unified L1 and Shared Memory

unit is more vulnerable to our attack, while the separate L1 and Shared Memory units can hide the timing leakage due to a Shared Memory bank conflict.

Our experimental results suggest that we could potentially exploit non-blocking execution mode to seal the timing leakage in the Shared Memory, while improving the overall performance of an application. We have observed that there are at most seven bank conflicts in our AES implementation due to the Shared Memory load instruction. Instead of stalling the execution pipeline at the next instruction that depends on the load result, we should execute other non-dependent instructions, such as one Global Memory load instruction or several *add* or *sub* instructions, until the Shared Memory load instruction completes. This technique can be implemented at the source code compilation stage. Thus, we could potentially hide the latency due to bank conflicts from a Shared Memory load instruction.

7 DISCUSSIONS AND COUNTERMEASURES

Although the penalty for each bank conflict is small, we are still able to exploit this fine-grained timing channel to recover confidential information on Kepler, Volta and Turing architectures. We choose to attack the AES encryption algorithm to demonstrate the feasibility and scalability of our attack. The attack should apply to other table-based cryptographic algorithms or even public-key ciphers on GPUs.

Although there have been a large number of countermeasures proposed to prevent timing attacks on CPUs [2, 4, 9, 17, 18, 20, 23, 24, 34, 38, 41], none of them can defend against our attack, because our attack exploits a very different timing channel than the common cache timing channel [11, 25, 37, 39]. More recently, RCoal [15] was proposed to defeat the timing attack using the memory coalescing unit on a GPU [13], but the countermeasure addresses the timing leakage only in the memory coalescing unit. Hence, it does not apply to our attack.

To thwart the attack described in this article, the countermeasure has to specifically target the Shared Memory banks. Avoiding using Shared Memory can prevent the attack, but would incur high performance degradation for some applications. However, eliminating Shared Memory bank conflicts can both improve an application's performance, as well as mitigate a differential timing attack. This can be done by reducing the Shared Memory data usage, such that the size of our data is no larger than 256 bytes (bank size $\times$ number of banks, 8×32). In this way, no bank conflicts would occur. For the AES encryption algorithm, we can change the AES implementation to use the SBox version. This implementation uses only a 256-byte table, spanning 32 banks. There will be no bank conflicts for 32 threads. However, an SBox-based implementation may not be as efficient as a T-table based implementation, and is not compatible with many existing cryptographic software libraries either. The security and performance of this implementation are demonstrated by Lin et al. [22].

As shown in the timing analysis on Maxwell and Pascal architectures in Section 6, we can leverage non-blocking execution mode to seal the timing leakage in Shared Memory. This can be done since the penalty for each bank conflict is only 2 GPU cycles. However, it cannot be done easily on the Kepler architecture, as it would require the Kepler to support independent and parallel instruction execution to hide more than 40 GPU cycles of delay caused by bank conflicts in the Shared Memory.

Another implementation technique called *scatter-gather* has been examined by Lin et al. [22] to prevent AES information leakage through the Shared Memory unit. This technique could also be applied to other table-based cryptographic algorithms. The technique requires modification to the table lookup procedure such that every Shared Memory access of a thread would touch all Shared Memory banks, and therefore, every Shared Memory access would result in a constant number of bank conflicts. This modification comes with some performance overhead. In the following, we

propose a multi-key implementation as an alternative approach that can effectively prevent our attack and does not need to modify the original implementation.

7.1 Multi-Key Implementation as Countermeasure

Using multiple keys during encryption has been deployed in commercial systems. By recovering one key, the adversary still cannot break the entire encryption system without knowing other keys. However, leveraging multiple keys during encryption will incur a much higher cost in secure storage, due to extra key management (mapping the keys to threads), and so will incur performance degradation.

So far in all our experiments, all threads in the encryption kernel are using the same key. This assumption significantly simplifies the attack strategy, as the attacker needs to guess only one key byte value to compute the shared memory bank conflicts among all 32 threads. In this section, we will evaluate the effectiveness of our attack against GPU AES implementations when multiple keys are being used. Specifically, we evaluate our attack when two, three, and four different keys are used in a block of threads. We consider two scenarios. In the first scenario, the attacker knows the mapping between encryption keys and GPU threads, while in the second scenario, the attacker does not know the key mapping. Under both scenarios, we demonstrate that the implementation that uses a few different keys is secure from our attack. All experiments are run on the Nvidia Kepler K40 GPU.

7.1.1 Encryption Key Mapping. Key mapping describes the way multiple keys are distributed to different blocks of input data. In our AES implementations, each GPU thread performs encryption on one block of data, and the block to GPU thread mapping is fixed and known. Thus, our job is to associate keys to threads. Note that we are working with symmetric block ciphers (e.g., AES), so the same key should be used to decrypt ciphertext on the receiver side, which increases the complexity of key management between the sender and receiver.

To map a key to a thread, we use the value of the thread id (within the block of threads) and compute the modulo of the total number of keys as the index to an array of keys. For example, when the implementation uses two different keys, threads with an even ID will be assigned the first key and threads with an odd ID will be assigned the second key.

7.1.2 Unknown Key Mapping. In these experiments, we assume the adversary does not know the key mapping, nor the number of different keys used during encryption. He/she can only apply the attack methodology as if all threads are using the same key as before. The result is shown in Figure 13.

From Figure 13, when two different keys are used, we can see that, without modifying the attack methodology, that these two key values are recovered in the 15th key byte. However, the DoMs for both key values are reduced from 17 (original implementation) to 3 GPU cycles. When attacking the 2-key implementation, we assume all of the threads are using the same key. Thus, we use the wrong key value to compute the associated access bank indices for 16 of 32 threads. However, if the number of bank conflicts is computed using 16 correctly computed bank indices, this will contribute to finding the correct key value. Otherwise, incorrect values will just contribute more noise. Thus, we see a significant drop in the DoMs when attacking the 2-key implementation.

As the total number of keys used in the implementation increases, the noise increases and the DoM value for the true key value decreases. When attacking the 4-key implementation, we cannot recover any key with 10 million samples, as shown in Figure 13.

We can always increase the number of samples to compensate for the noise, but it becomes impossible to attack the 32-key implementation. Recall that the number of bank conflicts falls into the range of [2, 7]. To correctly compute two bank conflicts, we need to know memory accesses

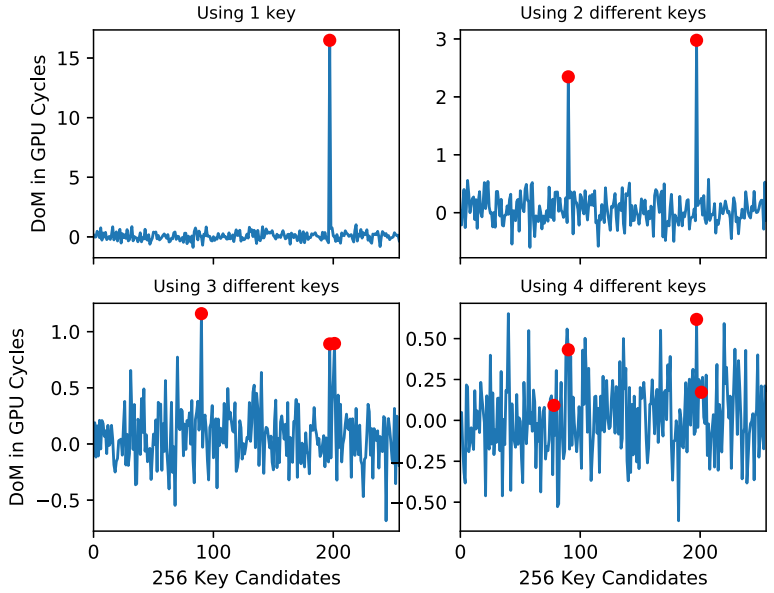


Fig. 13. The 15th key byte recovery using 10 million samples with GPU timing information.

from at least three threads. However, in the 32-key implementation, each thread has its own key, and we can only obtain the memory accesses by correctly guessing three key values assigned to those threads. By assuming all threads are using the same key, we cannot obtain the correct memory accesses, nor the number of bank conflicts in the 32-key implementation.

7.1.3 Known Key Mapping. In this experiment, we assume the adversary knows the total number of keys deployed in the implementation and the key mapping. The assumption allows us to guess multiple key byte values instead of one, and we can correctly compute the bank conflicts among all 32 threads, while the attack complexity will increase as we have to guess multiple bytes.

When attacking a 2-key implementation, we can retrieve two key values with a DoM value of 16 GPU cycles using 1 million samples. The result is what we have expected, since guessing two keys at the same time would allow us to compute the correct number of bank conflicts, which is the same as if we were attacking the 1-key implementation. However, guessing multiple key bytes at the same time increases the computational complexity exponentially. The complexity of attacking a 2-key implementation is 2^{16}. This approach will become computationally infeasible once the number of different keys used in the implementation reaches 5.

As we demonstrate in two scenarios, when 32 threads are using 32 different keys, it becomes impossible for our attack to succeed. Thus, deploying multiple different keys in encryption can help to prevent side-channel timing leakage.

7.1.4 Multi-Key Performance. While we show that multi-key implementations can resist our attack, multi-key implementations come with a performance penalty. When all threads are using a single key, the key can be loaded from memory in a single request and broadcast to all threads. When multiple keys are used, multiple memory requests are needed to load these keys. Thus, multi-key implementations will incur some performance degradation. We evaluate the performance overhead for implementations, varying the number of keys from 1 to 32. We use the performance of our 1-key implementation as the baseline, and compare others to it. The result is shown in Figure 14.

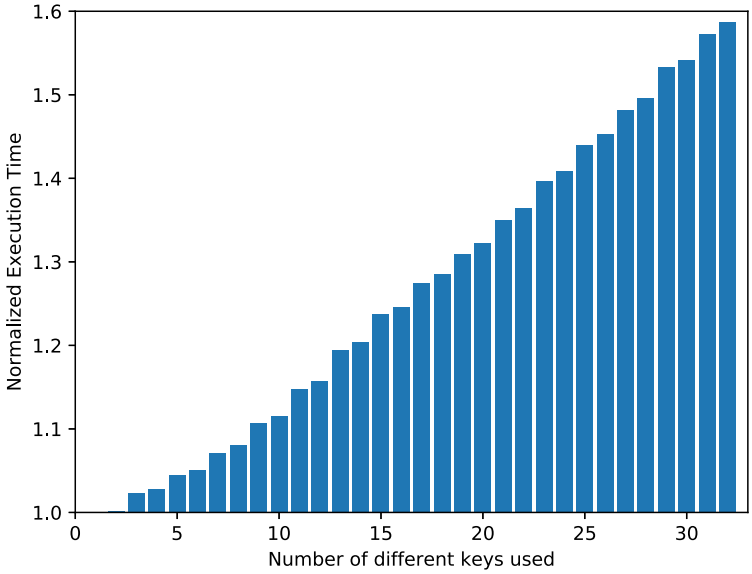


Fig. 14. Performance degradation when using multiple keys.

As the number of keys increases, the performance decreases. For the 32-key implementation, we see a performance degradation of 58% on the Kepler architecture and 48% on the Volta architecture. Although the multi-key implementation can be used to prevent our attack, we also need to consider the resulting performance impact.

8 CONCLUSION

In this article, we explore a new class of microarchitectural side-channel vulnerability in GPUs. Shared memory banking is used to improve the performance of memory access, but this same feature introduces a timing channel. We have developed a novel differential timing attack to exploit bank conflict-based timing channels, and for evaluation, we successfully recovered the encryption key for several GPU AES implementations. We anticipate our attack is applicable to other table-based cryptographic algorithms such as Blowfish. We investigate more realistic attack scenarios and quantify the effectiveness of our attack. We consider the attacks on Nvidia Kepler, Maxwell, Pascal, Volta and Turing devices. We also evaluate the effectiveness of multi-key implementations as a countermeasure and their associated execution time overhead.

In the future, we will focus on developing countermeasures at both the software and hardware levels, striving for minimal performance impact. As discussed in Section 6, it is possible to completely hide the timing leakage by exploiting non-blocking execution mode in GPUs. It would be interesting to investigate how the compiler could remove timing leakage in an application. We plan to explore this in future work.

REFERENCES

[1] Ahmed A. Abdelrahman, Mohamed M. Fouad, Hisham Dahshan, and Ahmed M. Mousa. 2017. High performance CUDA AES implementation: A quantitative performance analysis approach. In *Proceedings of the 2017 Computing Conference*. IEEE, 1077–1085.

[2] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. 2018. Leveraging hardware transactional memory for cache side-channel defenses. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, 601–608.

[3] Chonglei Mei, Hai Jiang, and J. Jenness. 2010. CUDA-based AES parallelization with fine-tuned GPU memory utilization. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. 1–7. DOI: https://doi.org/10.1109/IPDPSW.2010.5470766

[4] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Arch. Code Optimiz.* 8, 4 (2012), 35.

[5] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. 2014. QMS: Evaluating the side-channel resistance of masked software from source code. In *Proceedings of the 51st Annual Design Automation Conference*. 209:1–209:6.

[6] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. 2012. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes.

[7] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel. 2008. Mutual information analysis. In *Cryptographic Hardware and Embedded Systems (CHES'08)*. Springer Berlin Heidelberg, 426–442.

[8] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. 2012. GPU-acceleration of block ciphers in the OpenSSL cryptographic library. In *Proceedings of the International Conference on Information Security*. Springer, 338–353.

[9] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proceedings of the USENIX Security Symposium*.

[10] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: A fast and stealthy cache attack. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[11] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. A faster and more realistic flush+ reload attack on AES. In *Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 111–126.

[12] Z. H. Jiang and Y. Fei. 2017. A novel cache bank timing attack. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. 139–146. DOI: https://doi.org/10.1109/ICCAD.2017.8203771

[13] Z. H. Jiang, Y. Fei, and D. Kaeli. 2016. A complete key recovery timing attack on a GPU. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. DOI: https://doi.org/10.1109/HPCA.2016.7446081

[14] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A novel side-channel timing attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI'17)*. ACM, New York, NY, 167–172. DOI: https://doi.org/10.1145/3060403.3060462

[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] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2018. A timing side-channel attack on a mobile GPU. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'18)*. IEEE.

[17] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the USENIX Security Symposium*. 189–204.

[18] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 974–987.

[19] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Proceedings of the Annual International Cryptology Conference*. Springer, 388–397.

[20] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2009. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 393–404.

[21] Qinjian Li, Chengwen Zhong, Kaiyong Zhao, Xinxin Mei, and Xiaowen Chu. 2012. Implementation and analysis of AES encryption on GPU. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication and 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, 843–848.

[22] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. 2019. Scatter-and-gather revisited: High-performance side-channel-resistant AES on GPUs. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*. ACM, 2–11.

[23] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. IEEE, 406–418.

[24] Fangfei Liu and Ruby B. Lee. 2014. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 203–215.

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security & Privacy*.

[26] Chao Luo, Yunsi 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 (ICCAD'18)*. ACM, New York, NY, Article 113, 8 pages. DOI: https://doi.org/10.1145/3240765.3240812

[27] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. 2015. Side-channel power analysis of a GPU AES implementation. In *Proceedings of the 2015 33rd IEEE International Conference on Computer Design (ICCD'15)*. IEEE, 281–288.

[28] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on GPGPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*. ACM, New York, NY, 354–366. DOI: https://doi.org/10.1145/3123939.3124538

[29] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2139–2153.

[30] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. 2012. High-performance symmetric block ciphers on multicore CPU and GPUs. *Int. J. Netw. Comput.* 2, 2 (2012), 251–268.

[31] Naoki Nishikawa, Keisuke Iwai, Hidema Tanaka, and Takakazu Kurokawa. 2014. Throughput and power efficiency evaluation of block ciphers on kepler and gcn gpus using micro-benchmark analysis. *IEICE Trans. Inf. Syst.* 97, 6 (2014), 1506–1515.

[32] Nvidia. 2015. Nvidia CUDA Toolkit v7.0 Documentation. Retrieved from http://docs.nvidia.com/cuda/index.html.

[33] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. 2010. Fast software AES encryption. In *Proceedings of the International Workshop on Fast Software Encryption*. Springer, 75–93.

[34] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. ACM, 77–84.

[35] Matthieu Rivain. 2009. On the exact success rate of side channel analysis in the Gaussian model. In *Selected Areas in Cryptography*. 165–183.

[36] Bruce Schneier. 1993. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *Proceedings of the International Workshop on Fast Software Encryption*. Springer, 191–204.

[37] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71.

[38] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[39] Yuval Yarom and Katrina Falkner. 2014. Flush+ reload: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the USENIX Security Symposium*. 719–732.

[40] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. Cachebleed: A timing attack on OpenSSL constant time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.

[41] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 871–882.

Received February 2019; revised July 2019; accepted September 2019