A Novel Side-Channel Timing Attack on GPUs

Zhen Hang Jiang, Yunsi Fei, David Kaeli
Electrical & Computer Engineering Department, Northeastern University
Boston, MA 02115, USA
zjiang@ece.neu.edu, yfei@ece.neu.edu, kaeli@ece.neu.edu

ABSTRACT

To avoid information leakage during program execution, modern software implementations of cryptographic algorithms target constant timing complexity, i.e., the number of instructions executed does not vary with different inputs. However, many times the underlying microarchitecture behaves differently when processing varying data inputs, which covertly leaks confidential information through the timing channel. In this paper, we exploit a novel fine-grained microarchitectural timing channel, stalls that occur due to bank conflicts in a GPU's shared memory. Using this attack surface, we develop 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 AES encryption key using 10 million samples. We also evaluate the scalability of our attack method by attacking a 8192-thread implementation of the AES encryption algorithm, recovering some key bytes using 1 million samples.

1. INTRODUCTION

Originally, GPU devices were solely designed to perform efficient graphic rendering. Since the birth of programmable shader cores and high-level programming language frameworks [2, 7], GPU devices have assumed a major role in a range of parallel computing environments. Cloud-based services, such as encryption/decryption of large data sets stored on disks, can effectively exploit the parallelism found on GPU devices to deliver high throughput. Earlier work by Jiang et al. [3] explored the vulnerability of running encryption algorithms on GPU devices to side-channel timing attacks, despite the fact that the encryption algorithm is mathematically proven to be secure. In this paper, we identify another timing side-channel that can be utilized to reveal sensitive information. Specifically, we develop a differential timing attack that exploits timing variability due to shared memory bank conflicts. We exploit this microarchitectural feature to recover all AES encryption key bytes.

GLSVLSI '17, May 10-12, 2017, Banff, AB, Canada
© 2017 ACM. ISBN 978-1-4503-4972-7/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3060403.3060462

The GPU on-chip shared memory is an important hardware unit for alleviating memory traffic to off-chip device memory. It is designed to serve data that is shared and frequently accessed by many running threads. In order to optimize Single Instruction Multiple Thread (SIMT) execution on the GPU, the shared memory needs to provide high memory throughput. Thus, in modern GPUs a shared memory is divided into multiple memory banks, versus a monolithic bank. With multiple shared memory banks, multiple memory access requests for different memory banks can be served in parallel, therefore significantly improving overall memory throughput.

However, when multiple memory requests access the same bank, they are serviced in a serial fashion. As a result, the time to service multiple memory requests that reside in same bank is noticeably longer as compared to accesses that reside in different banks. This has been studied in terms of tuning memory performance, but has only recently been considered as a potential vulnerability. Yarom et al. investigate how the sensitive information can be leaked when a public cipher is running on CPUs with multi-bank caches [9]. A GPU presents a much more complex microarchitecture, especially when we consider the degree of parallelism on a GPU, as well the non-deterministic behavior of the hardware-based scheduler. In this paper, we carefully examine this timing channel in GPUs and develop an effective differential timing attack to retrieve the secret key.

The paper is organized as follows: In Section 2, we provide background on the Advanced Encryption Standard (AES) algorithm, as well as touch on the GPU memory hierarchy and execution model. In Section 3, we explore timing variations present due to shared memory bank conflicts, i.e., identifying the memory bank timing channel. In Section 4, we describe our differential timing attack targeting table lookup-based cryptographic algorithms, and attack AES encryption. In Section 5, we discuss feasible countermeasures to prevent the attack. Finally, we conclude the paper in Section 6.

2. BACKGROUND

In this section, we describe the AES implementation is evaluated on the targeted GPU platform, as well as the memory hierarchy and execution model of Nvidia's Kepler GPU architecture [7].

2.1 AES Encryption

In this paper, we evaluate the timing leakage vulnerability of a table-based cryptographic algorithm on a GPU. We use

128-bit ECB mode AES encryption as an example. The implementation was ported from the OpenSSL 0.9.7 library into CUDA code, which is similar to the implementation used in [3]. However, instead of storing lookup tables in the global memory, we store them in the shared memory to demonstrate the timing leakage in the shared memory. We transform the AES encryption procedure into a single GPU kernel, where each GPU thread can process 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, and the last round omitting the MixColumni operation. For faster processing, the first three operations are integrated into T-table lookups in the first nine rounds, and in the last round, a special T-table ($Te4$) lookup is performed, integrating the SubByte with the ShiftRow operation.

Each encryption round requires one 16-byte round key. Ten round keys are generated by the key scheduler using one 16-byte user-specified secret key during encryption. Knowing any round key, an attacker can compute the original 16-byte secret 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.

Listing 1: AES Last Round Code Snippet

```
s0 = (Te4[(t0 >> 24)       ] & 0xff000000)^
     (Te4[(t1 >> 16) & 0xff] & 0x00ff0000)^
     (Te4[(t2 >>  8) & 0xff] & 0x0000ff00)^
     (Te4[(t3      ) & 0xff] & 0x000000ff)^
     k0;
```

Variable $s0$ is the first four bytes of the 16-byte ciphertext. Each of variables, t0, t1, ..., t3, contains 4 bytes of the input state for the last round. A selected byte of each variable is used to look up a T-table to obtain a four-byte output, of which only one byte contributes to the final ciphertext. k_0 is the first four bytes of the last round key. From the original algorithm, the last-round can be simplified as byte-wise operations shown below:

$$c_j = SBox[s_i^9] \oplus rk_j \tag{1}$$

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 9_{th}-round output state, s_i^9, can be calculated once the corresponding cipher and key byte are known by:

$$s_i^9 = SBox^{-1}[c_j \oplus rk_j] \tag{2}$$

2.2 Nvidia GPU Memory Hierarchy

Our testing platform is equipped with an Nvidia Tesla K40 GPU. There is an off-chip DRAM memory (device memory), and it 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 (with 192 single-precision cores) also has L1, texture, and constant caches. Data in those caches are private to threads running in the SMX. In addition, there is a shared memory in each SMX, and only the block of threads that allocate the data in the shared memory can access them. Also, each GPU thread owns an exclusive set of 255 registers to store its current state.

The GPU shared memory and L1 cache reside in the same physical memory storage. On the K40 GPU, the total size is 64 KB. The individual size of the shared memory and L1 cache can be configured. In our case, we allocate 48 KBs of space as shared memory and 16 KB for the L1 cache. Note that the size configuration does not affect the results presented in this paper.

The shared memory is divided into 32 banks, and it has a configurable bank line size (annotated as *banksize* in the Nvidia documentation [7]): 4 bytes or 8 bytes. For evaluating the AES algorithm, we use a bank size of 8 bytes, enabling faster encryptions. The memory address breakdown for shared memory is shown in Figure. 1, where the three least-significant bits are used for the bank offset, and the next five bits (bit 3-7) are the bank index and are used for selecting a bank.

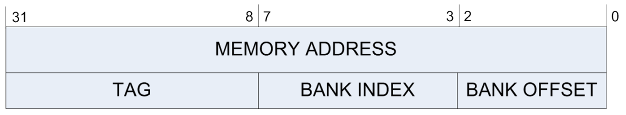

Figure 1: Memory address to shared memory bank mapping

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 single bank design. On the other hand, when two memory requests (with the same bank index, but different tag values) are accessing the same bank, they will be handled in a serialized fashion, which we refer to as a *bank conflict*. 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, each thread within a warp must be synced 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 a memory instruction, each thread will generate one memory request. A warp of threads will generate 32 memory requests for one memory instruction. Across the accesses, any duplicate memory requests will be merged and the distinct addresses are sent to the shared memory. Under the SIMT model, the execution time of this memory instruction will be determined by the memory bank that receives the highest number of bank conflicts (the largest number of requests resolving to the same bank). In other words, the execution time of a shared memory instruction becomes highly dependent on the memory address issued, and any resulting bank conflicts. We can use these differences to identify when the GPU is potentially leaking secret information.

3. CACHE BANK CONFLICTS

In this section, we conduct experiments to examine the impact of bank conflicts on the program execution time. We develop a kernel that uses a warp of threads to perform loads from the shared memory. Depending on the address of data that each thread accesses, a number of bank conflicts occur, resulting in different execution time of the load operation.

We deliberately create a specific number of bank conflicts by selecting the address that each thread is accessing. Using a high resolution time-stamping mechanism, we can study the impact of bank conflicts on kernel execution time. We have developed the micro-benchmark shown in Listing 2.

Listing 2: Micro-Benchmark 1

```
register uint32_t tmp,tmp2,offset = 64;
__shared__ uint32_t share_data[1024*4];
...
int tid=blockDim.x*blockIdx.x+threadIdx.x;
tmp = clock();
tmp2  = share_data[tid*stride+0*offset];
tmp2 += share_data[tid*stride+1*offset];
...
tmp2 += share_data[tid*stride+39*offset];
times[tid] = clock() - tmp;
in[tid] = tmp2;
```

The purpose of the microbenchmark is to measure the time for a sequence of memory accesses of each thread in a block. In Listing 2, the variable *share_data* points to a continuous 16KB memory space in shared memory. Each thread is accessing 40 memory locations in a sequence, with an offset of 64 words between two adjacent addresses. From Figure. 1, we know that two adjacent memory addresses have the same bank index and bank offset (64 words = 2^8 bytes), and therefore all the memory addresses requested by a single thread are on the same bank. Each thread accesses one memory region, and the offset between memory regions by different threads is one or multiple *strides*. By manipulating the value of *stride*, we can intentionally create bank conflicts among threads in a block. We run this kernel 10,000 times and collect 10,000 timing information for each stride value, ranging from 1 to 32 word. The timing distribution for 32 different stride values is shown in Figure. 2.

We observe that there are only 5 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, shared memory is divided into 32 banks, the bank size is configured to be 8 bytes, and the memory address breakdown is shown in Figure 1. Given a word index for the *shared_data* array, we can calculate the bank index by dropping the first least-significant bit and then module 32, 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 modulus operation.

As an example, assuming a stride value 2, for a memory access instruction issued across a warp of 32 threads, we will generate the following 32 memory indices: {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62}. By using Equation (3), we have the following bank access indices for the warp: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31}. Thus, there is no bank conflicts produced when the stride is 2. 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. 2. Each group corresponds to a different number of bank conflicts and average timing.

We also plot the average execution time for a group (for selected stride values) versus the number of shared memory bank conflicts in Figure. 2. We can easily identify a linear relationship. The slope of the linear line is 392, with an offset of 1002 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 bank conflicts is not as large as the cache miss penalty (another well-studied cache side-channel), we will show the feasibility of exploiting this fine-grained timing channel for key retrieval through statistical methods.

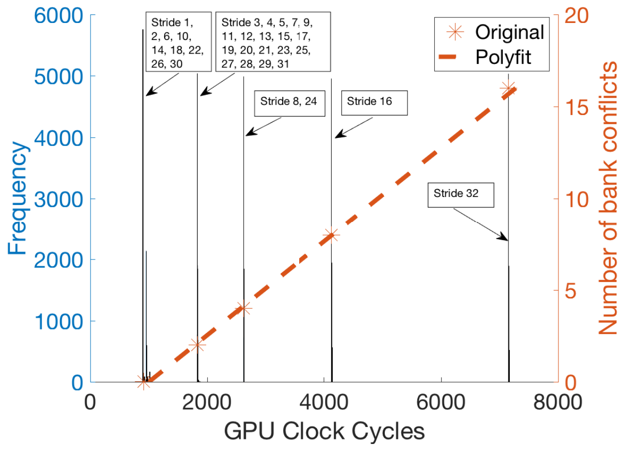

Figure 2: Number of bank conflicts vs timing for 32 stride values

4. DIFFERENTIAL TIMING ATTACK

In this section, we apply our differential timing analysis method 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 an entire encryption is also dependent on the number of bank conflicts created by the table lookup operations.

Since the index for table lookup operations is related to the round key, with the correct key guess, we can predict the number of bank conflicts during one round of AES operation 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 tend to be low. This is the basic principle for differential timing attack, similar to the traditional differential power attack (DPA).

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.

4.1 Mapping Between the AES Lookup Tables and GPU Shared Memory Banks

As we describe in Section 2, the AES implementation we use contains 5 lookup tables, which are placed one after another and occupy a contiguous memory space in the shared memory. Since we are attacking the last round of AES encryption, we only need to examine the *Te4* lookup table 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 *Te4* lookup table. The same Equation (3) can be used to calculate the bank index, given a lookup table index.

4.2 Collecting Data

The data collection procedure is similar to the experiments that we performed in Section 3. Instead of 40 shared 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 is composed of 32 16-byte ciphertexts and a timing sample, as shown in the following format:

$$[\{C^0, C^1, ..., C^{15}\}, t]$$

$$where\ each\ C^i\ is\ a\ 16-byte\ ciphertext\ produced\ by$$

$$thread\ i,\ consisting\ of\ 16\ bytes\ \{c_0^i, c_1^i, ..., c_{15}^i\},$$

$$and\ t\ is\ the\ encryption\ time\ for\ this\ warp$$

In this paper, we consider the encryption time measured both from the GPU and CPU. The encryption time measured from the GPU side provides more accurate measurements of the encryption, while measurements on the CPU side contain more noise sources because of the the non-deterministic data transfer times between the GPU and the CPU, as well as other required initialization procedures in the GPU device for running a kernel.

4.3 Calculating the Shared Memory Bank Index

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

$$\{s_i^0, s_i^1, ..., s_i^{31}\} = SBox^{-1}[\{c_j^0, c_j^1, ..., c_j^{31}\} \oplus rk_j] \qquad (4)$$

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 j^{th} last round key byte, which is common to all the threads. These lookup table indices, $\{s_i^0, s_i^1, ..., s_i^{31}\}$, are exactly the shared memory indices. We use Equation (3) to calculate the bank indices used by all threads in a warp for this table lookup instruction, and can derive the number of bank conflicts.

4.4 Recovering Key Bytes

We can launch a correlation timing attack (CTA) using the collected data. As shown in Listing 1, the code snippet for the last round AES, each T-table lookup uses one state byte, 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 j^{th} last round key byte. For each 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. This is called CTA. For the data collected, the number of bank conflicts between the 32 threads falls in the range of [2, 4].

The power of CTA is determined by 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 CTA may not be more effective than differential timing attack (DTA), which only considers two points. We adopt a DTA approach, and calculate two average timing values, one for the bin of data samples that generate two bank conflicts, and the other for the bin of data samples that generate four bank conflicts. The difference-of-means (DoMs) between them should be about two times the penalty for one bank conflict, i.e., around 19 cycles. We first apply the attack method to the 15^{th} key byte, and the result is shown in Figure. 3 (the upper plot).

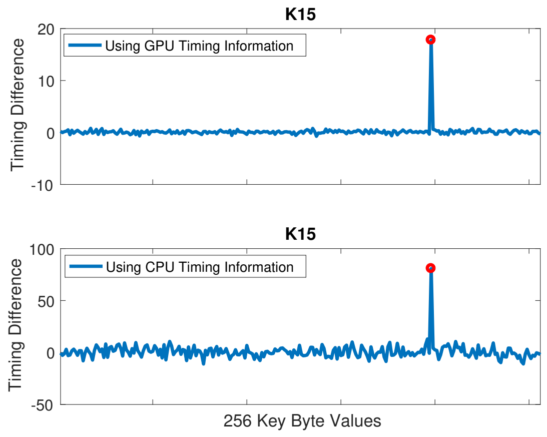

Figure 3: 15^{th} key byte recovery using 10 million samples with both GPU and CPU timing information.

The correct key byte value (198) is highlighted in red. The timing difference for the true key value is 17.75 GPU cycles, which is about 2 GPU cycles less than the predicted signal, 19.6 GPU cycles, and the timing difference for other values is much smaller, between -2.3 to 2.3 GPU cycles. By increasing the sample size to 10 million, we found that the timing difference for the true value remains the same, while the timing differences for wrong values are reduced to a range between -0.8 and 0.8 GPU cycles.

We apply this attack methodology and recover other key bytes. The result is shown in Figure. 4. 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 difference, e.g., k_{15}. The reduced signal in k_0 is probably caused by instruction-level parallelism. Although the architectural details of the Nvidia Kelper GPU are not public, we suspect that the Kepler GPU can continue issuing independent instruction(s) in each cycle until all resources are being used up, and therefore, it hides the latency due to shared memory bank conflicts by issuing and executing other independent instructions.

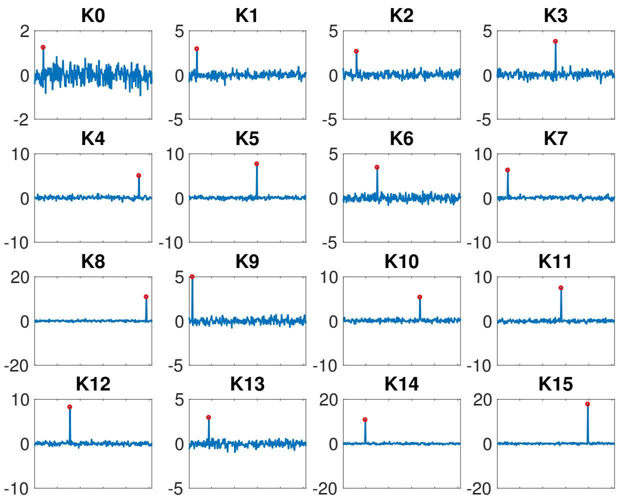

Figure 4: All 16 key bytes recovery using 10 million samples

To gain more insights into the GPU internals and resulting timings, we slightly modify Microbenchmark 1, as shown in Listing 3, to make each load instruction and the addition instruction independent from each other:

Listing 3: Micro-Benchmark 2

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

We run Micro-benchmark 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 similar linear relationship shown in Figure. 2. However, the slope and offset are 177 and 368 GPU cycles, respectively, and therefore the per-conflict penalty is reduced to 4.4 from 9.8 GPU cycles in Micro-benchmark 2.

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 Micro-benchmark 2 is similar to a non-blocking execution mode of Micro-benchmark 1, because all the loads are independent and therefore can be issued in an out-of-order manner. Listing 5 shows the SASS code for the modified benchmark, which is a sequence of independent loadings. While Micro-benchmark 1 introduces blocking, as shown in the original SASS code in Listing 4, where the loaded data is used by another operation that is 3 instructions away from the load operation. The read-after-write (RAW) data dependencies prevent later load instructions from being executed earlier.

Listing 4: Original SASS Code

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

Listing 5: Modified SASS Code

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

For our AES algorithm implementation, each lookup operation in the last round (16 lookups in total) is independent from each other. Since k_0 is being used in the first lookup operation, its delay due to bank conflicts is diminished by executing other independent lookup operations. Thus, we see a weaker signal in the k_0. While k_{15} is the last one to be processed, and the delay due to its bank conflict is completely exposed in the execution time, i.e., k_{15} has the strongest signal.

4.5 More Threads and Noise Sources

In the previous section, we have been using timing information taken from the GPU side, which contains much less noise than the timing information measured from the CPU side. Also, we evaluated our attack method only on a 32-thread AES encryption. It is important to understand the scalability of the attack (i.e., how the number of traces needed grows as we increase the number of threads). In this section, we evaluate the effectiveness of the attack using the CPU timing information and on implementations with higher numbers of threads, which better reflect typical scenarios for GPU execution.

4.5.1 *Using the 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, it has a higher resolution (4.8 times higher). However, the CPU timing information is much more noisy than the GPU timing information. We perform the same differential timing attack against our 32-thread implementation, and we can still recover key bytes, but with a weaker signal, as shown in Figure. 3 (the lower plot).

Using the CPU timing information, the timing difference is around 80.75 CPU cycles (16.83 in GPU cycles) between the two bins for the correct key guess (which is slightly lower than using the GPU timing information). The standard deviation of the timing difference using CPU timing is larger than that using GPU timing (6.7 vs 1.1). This implies that CPU timing contains more noise than GPU timing, and more data samples would be needed to achieve the same attack result as using GPU timing.

4.5.2 *Higher Number Threads Implementation*

In this section, we evaluate the scalability of our attack by attacking an 8192-thread AES implementation using 16 blocks of 512 threads (each block contains 16 warps), which is several times more than the maximum number (2880) of threads that can run in parallel on the Kepler K40 GPU.

Note, in such a real attack scenario, the attackers cannot manipulate the kernel code running on GPUs (they cannot easily insert time stamps before and after the encryption). Instead, they would typically 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, and we cannot know 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. We anticipate much high noise contained in this attack model, because the selected warp may not run on the slowest SMX, and also even if it does, there are other warps competing for the same SMX resources. Attributing 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 threads register states, can also contribute to the noise.

We use exactly the same attack method described before. We collect 1 million samples (rather than 10 million due to its excessively slow execution) of the 8192-thread AES encryption, and we apply our differential timing attack to recover the 15^{th} key byte. The result is shown in Figure. 5. The timing difference for the 15^{th} true key value is 94.2 CPU cycles, but it has a standard deviation of 16.8, which indicates there is a high degree of noise in the data samples that we collected.

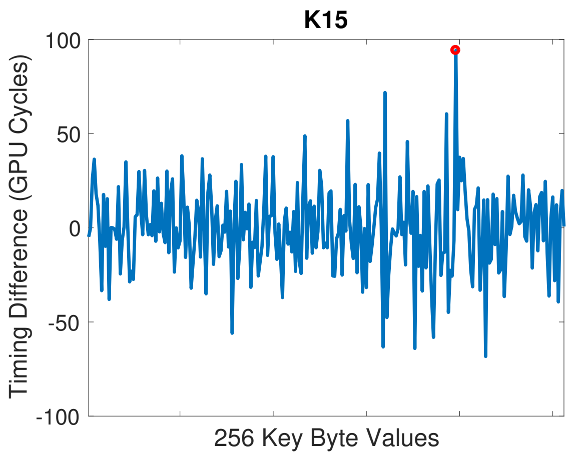

Figure 5: 15^{th} **key byte recovery using 1 million samples**

5. DISCUSSION

Although the penalty for each bank conflict is small, we are still able to exploit this fine-grained timing channel to recover confidential information. We choose to attack the AES encryption algorithm to demonstrate the feasibility and scalability of our attack. The attack should be applicable to other table-based cryptographic algorithms or even public ciphers on GPUs, as long as there are sensitive data-dependent shared memory accesses in the cryptosystem.

Our attack exploits the timing difference due to shared memory bank conflicts. Avoiding using shared memory unit can prevent the attack, but would incur performance degradation for some applications. However, removing shared memory bank conflicts can improve an application's performance as well as mitigating 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 * number of banks, $8 * 32$). In this way, no bank conflict can happen. For the AES encryption algorithm, we can revert the AES implementation back to the SBox version. Such 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 one.

Although there have been a large number of countermeasures proposed to prevent timing attacks on CPUs [5, 4, 1], to our knowledge, we have not found any of them that can be used to avoid our attack, since our attack exploits a different timing channel than the cache timing channel [8, 6]. As more research is carried out in the side-channel attack field, more timing channels will be found and exploited. Thus, we need new countermeasures against such newly found timing channels in order to prevent those unexpected attacks.

6. CONCLUSION

In this paper, we expose a new class of side-channel vulnerability in GPUs. We exploit the relationship between the AES execution time and the number of bank conflicts. We construct a differential timing attack on the GPU AES implementation to retrieve the key. Our attack is shown to effectively exploit this fine-grained timing channel. We also demonstrate that our attack can scale well. Future work includes developing countermeasures at multiple hardware and software levels.

Acknowledgment

This work was supported in part by National Science Foundation under grants STARSS-1618379 and MRI-1337854.

7. REFERENCES

[1] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[2] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes, 2012.

[3] Z. H. Jiang, Y. Fei, and D. Kaeli. A complete key recovery timing attack on a gpu. In *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, March 2016.

[4] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE Int. Symp. on High Performance Computer Architecture*, pages 393–404, 2009.

[5] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE Int. Symp. on High Performance Computer Architecture*, pages 406–418. IEEE, 2016.

[6] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symp. on Security & Privacy*, 2015.

[7] Nvidia. Nvidia cuda toolkit v7.0 documentation, 2015.

[8] Y. Yarom and K. Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symp.*, pages 719–732, 2014.

[9] Y. Yarom, D. Genkin, and N. Heninger. Cachebleed: A timing attack on OpenSSL constant time RSA, Aug. 2016.