A Complete Key Recovery Timing Attack on a GPU

Zhen Hang Jiang, Yunsi Fei, David Kaeli
Electrical & Computer Engineering Department, Northeastern University
Boston, MA 02115 USA

ABSTRACT

Graphics Processing Units (GPUs) have become mainstream parallel computing devices. They are deployed on diverse platforms, and an increasing number of applications have been moved to GPUs to exploit their massive parallel computational resources. GPUs are starting to be used for security services, where high-volume data is encrypted to ensure integrity and confidentiality. However, the security of GPUs has only begun to receive attention. Issues such as side-channel vulnerability have not been addressed.

The goal of this paper is to evaluate the side-channel security of GPUs and demonstrate a complete AES (Advanced Encryption Standard) key recovery using known ciphertext through a timing channel. To the best of our knowledge, this is the first work that clearly demonstrates the vulnerability of a commercial GPU architecture to side-channel timing attacks. Specifically, for AES-128, we have been able to recover all key bytes utilizing a timing side channel in under 30 minutes.

1. INTRODUCTION

With introduction of programmable shader cores and high-level programming frameworks [1, 2], GPUs have become fully programmable parallel computing devices. Compared to modern multi-core CPUs, a GPU can deliver significantly higher throughput by executing workloads in parallel over thousands of cores. As a result, GPUs have quickly become the accelerator of choice for a large number of applications, including physics simulation, biomedical analytics and signal processing. Given their ability to provide high throughput and efficiency, GPUs are now being leveraged to offload cryptographic workloads from CPUs [3, 4, 5, 6, 7, 8]. This move to the GPU provides cryptographic processing to achieve up to 28X higher throughput [3].

While an increasing number of security systems are deploying GPUs, the security of GPU execution has not been well studied. Pietro et al. identified that information leakage can occur throughout the memory hierarchy due to the lack of memory-zeroing operations on a GPU [9]. Previous work has also identified vulnerability of GPUs using software methods [10] [11]. While there has been a large number of studies on side-channel security on other platforms, such as CPUs and FPGAs, there has been little attention paid to side-channel vulnerability of GPU devices.

Timing attacks have been demonstrated to be one of the most powerful classes of side-channel attacks [12, 13, 14, 15, 16, 17]. Timing attacks exploit the relationship between input data and the time (i.e., number of cycles) for the system to process/access the data. For example, in a cache collision attack [13], the attacker exploits the difference in terms of CPU cycles to serve a cache miss versus a hit, and considers the cache locality produced by a unique input data set. There is no prior work of evaluating timing attacks on GPUs. To the best of our knowledge, our work is the first one to consider timing attacks deployed at the architecture level on a GPU.

The GPU's Single Instruction Multiple Threads (SIMT) execution model prevents us from simply leveraging prior timing attack methods adopted for CPUs on GPUs. A GPU can perform multiple encryptions concurrently, and each encryption will compete for hardware resources with other threads, providing the attacker with confusing timing information. Also, when using SIMT, the attacker would not be able to time-stamp each encryption individually. The timing information that the attacker obtains would be dominated by the longest running encryption. Given these challenges in GPU architectures, most existing timing attack methods become infeasible.

In this paper, we demonstrate that information leakage can be extracted when executing on an SIMT-based GPU to fully recover the encryption secret key. Specifically, we first observe that the kernel execution time is linearly proportional to the number of unique cache line requests generated during a kernel execution. In the L1-Cache memory controller in a GPU, memory requests are queued and processed in the First-In-First-Out (FIFO) order, so the time to process all of memory requests depends on the number of memory requests. As AES encryption generates memory requests to load its S-box/T-tables entries, the addresses of memory requests are dependent on the input data and encryption key. Thus, the execution time of an encryption kernel is correlated with the key. By leveraging this relationship, we can recover all the 16 AES secret key bytes on an Nvidia Kepler GPU. Although we demonstrate this attack on a specific Nvidia GPU, other GPUs also have the same exploitable leakage.

We have set up a client-server infrastructure shown in Figure. 1. In this setting, the attacker (client) sends

messages to the victim (encryption server) through the internet, the server employs its GPU for encryption, and sends encrypted messages back to the attacker. For each message, the encrypted ciphertext is known to the attacker, as well as the timing information. If the timing data measured is clean (mostly attributes to the GPU kernel computation), we are able to recover all the 16 key bytes using one million timing samples. In a more practical attack setting where there is CPU noise in our timing data, we are still able to fully recover all the key bytes by collecting a larger number of samples and filtering out the noise. Our attack results show that modern SIMT-based GPU architectures are vulnerable to timing side-channel attacks.

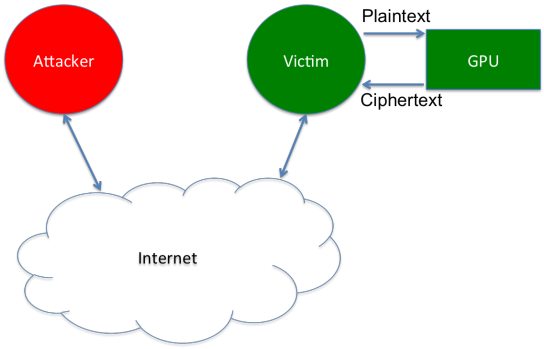

Figure 1: The attack environment

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we provide an overview of our target GPU memory architecture and our AES GPU implementation. In Section 4, the architecture leakage model is first presented, followed by our attack method that exploits the leakage for complete AES key recovery. We discuss potential countermeasures in Section 5. Finally, the paper is concluded in Section 6.

2. RELATED WORK

Timing attacks utilize the relationship between data and the time taken in a system to access/process the data. Multiple attacks have been demonstrated successfully by exploiting cache access latencies, which leak secrets through generating either cache contention or cache reuse [13, 14, 15]. In order to create cache contention or cache reuse, attackers need to have their own process(spy process) coexisting with the targeted process (victim process) on the same physical machine. This way the spy process can evict or reuse cache contents created by the victim process to introduce different cache access latencies. We refer to this class of attacks as *offensive attacks*. Another kind, *non-offensive attack*, has also been demonstrated successfully by Bernstein [12]. Unlike *offensive attacks*, Bernstein's timing attack does not interfere with the victim process. His attack exploits the relationship between the time for an array lookup and the array index.

The attack strategy commonly deployed in CPU-based timing attack methods consist of one block of ciphertext, and profiling the associated time to process that block. However, on a GPU it would be highly inefficient to only perform one block encryption and produce one block of ciphertext at a time, given the GPU's massive computational resources. In a real world scenario, the encryption workload would contain multi-block messages, and on each data sample, the GPU timing attackwould produce many blocks of ciphertext. The key difference is that the GPU scenario will only collect a single timing value for the multiple blocks. Although many successful attack methods have been demonstrated on CPU platforms, these methods cannot be directly applied to the GPU platform due to a lack of accurate timing information and nondeterminism in thread scheduling. Our timing attack method targets a GPU and is non-offensive like Bernstein's. We exploit the inherent parallelism present on the GPU, as well as its memory behavior, in order to recover the secret key.

3. BACKGROUND

In this Section, we discuss the memory hierarchy and memory handling features of Nvidia's Kepler GPU architecture [18]. Note, not all details of the Kepler memory system are publicly available - we leverage information that has been provided by Nvidia, as well as many details of the microarchitecture we have been able to reverse engineer. We also describe the AES implementation we have evaluated on the Kepler and the configuration of the target hardware platform used in this work.

3.1 GPU Memory Architecture

3.1.1 Target Hardware Platform

Our encryption server is equipped with an Nvidia Tesla K40 GPU. The Kepler-family device includes 15 streaming multiprocessors (SMXs). Each SMX has 192 single-precision CUDA cores, 64 double-precision units, 32 special function units, and 32 load/store units (LSU). In CUDA terminology, a group of 32 threads are called a *warp*. Each SMX has four warp schedulers and eight instruction dispatch units, which means four warps can be scheduled and executed concurrently, and each warp can issue two independent instructions in one GPU cycle [18].

3.1.2 GPU Memory Hierarchy

Kepler provides an integrated off-chip DRAM memory, called *device memory*. The CPU transfers data to and from the device memory before and after it launches the kernel. Global memory, texture memory, and constant memory reside in the device memory. Data residing in the device memory is shared among all of the SMXs. Each SMX has L1/shared, texture, and constant caches, and they are used to cache data from global memory, texture memory, and constant memory, respectively. These caches are placed very close to the physical cores, so they have much lower latency than

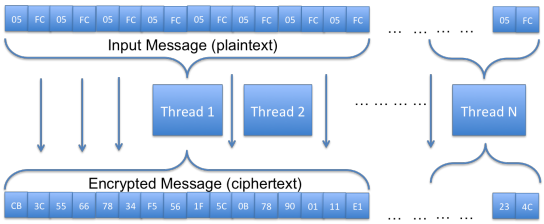

Figure 2: The GPU AES implementation used in this work.

the corresponding memories. Texture memory/cache are optimized for spatial memory access patterns. Constant memory/cache is designed for broadcasting a single constant value to all threads in a warp. Global memory, together with the L1 and L2 caches and coalescing units (units that can coalesce multiple global memory accesses from a warp into memory transactions) provide for fast general-purpose memory accesses. The hierarchy of the L1/L2 caches and global memory is similar to that found on the modern multi-core CPUs. The sizes of the L1 and L2 caches on a GPU are much smaller that those found on a CPU. However, GPU caches have much higher bandwidth, which is needed to support a large number of cores.

3.1.3 *Memory Request Handling*

On a Kepler GPU, a load instruction for a warp generates 32 memory requests if none of the threads is masked. All 32 memory requests are sent to coalescing units, which reorders pending memory accesses, trying to reduce memory requests down to a number of unique cache line requests. These cache line requests are issued to L1-cache controller, one per cycle, and the process is referred to as *memory issue serialization* [19]. If the requested data is present in the L1-cache, the data is loaded into the specified register and the cache line request is resolved in one GPU cycle by the LSU. On a miss, the request is queued in a MSHR (Miss Status Holding Register), one per cycle. If any incoming cache line request matches an outstanding cache line miss queued in the MSHR, the request is merged into a single entry in the MSHR. All requests queued in the MSHR are processed in FIFO order. These requests are then forwarded to the next level memory controllers (L2 or device memory). Upon receiving the requested data, the LSU needs to load this data into the register file and release the MSHR entry, one per cycle. This process is referred to as *writeback serialization* [19].

3.2 AES GPU Implementation

In this paper, we evaluate 128-bit Electronic Codebook (ECB) mode AES encryption based on T-tables, which operates on blocks of 16 byte data, using a secret key of 16 bytes. The encryption implementation we use was ported from OpenSSL 0.9.7 library into CUDA. We transformed an entire block encryption into a single GPU kernel, so that each thread in the GPU can process one block encryption independently, as shown in Figure. 2. The encryption key scheduling step expands the 16-byte secret key into 160-byte round keys for ten rounds of operation. In the initial round, the 16-byte plain text is XORed with the first round key to generate the initial state. In the T-table version of AES, SubByte, ShiftRow and MixColumn operations are integrated to perform lookups on four T-tables. Rounds 1-9 simply need to perform T-table lookups and add round keys. In the last round, a special T-table only integrates the SubByte with the ShiftRow and does not involve a MixColumn operation.

Our attack is focused on the last round of AES, whose operations are shown in Equation (1), where each T table lookup returns a 4-byte value that is indexed by a one byte value, c_{0-15} are the output cihpertext bytes and $\{t_0, t_1, \cdots, t_{15}\}$ are the input bytes to the last round.

$$\begin{aligned}
c_0 &= T4[t_3]_0 \oplus k_0 \\
c_1 &= T4[t_6]_1 \oplus k_1 \\
c_2 &= T4[t_9]_2 \oplus k_2 \\
c_3 &= T4[t_{12}]_3 \oplus k_3 \\
c_4 &= T4[t_7]_0 \oplus k_4 \\
c_5 &= T4[t_{10}]_1 \oplus k_5 \\
c_6 &= T4[t_{13}]_2 \oplus k_6 \\
c_7 &= T4[t_0]_3 \oplus k_7 \\
c_8 &= T4[t_{11}]_0 \oplus k_8 \\
c_9 &= T4[t_{14}]_1 \oplus k_9 \\
c_{10} &= T4[t_1]_2 \oplus k_{10} \\
c_{11} &= T4[t_4]_3 \oplus k_{11} \\
c_{12} &= T4[t_{15}]_0 \oplus k_{12} \\
c_{13} &= T4[t_2]_1 \oplus k_{13} \\
c_{14} &= T4[t_5]_2 \oplus k_{14} \\
c_{15} &= T4[t_8]_3 \oplus k_{15}
\end{aligned} \tag{1}$$

Each generation of a ciphertext byte involves a table lookup (which returns a 4-byte value), byte positioning (taking one byte out of four bytes done by byte masking and shifting), and add-key. When implemented on Nvidia GPUs, each generation of a ciphertext byte will be implemented in CUDA using load and store instructions, in addition to logic instructions. Although the order of table lookups for the cipher bytes is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, the order of CUDA load and store instructions for all cipher bytes may be different depending on how the program is compiled. For example, the CUDA compiler, *nvcc*, by default assume $-O3$ optimization is enabled, which reorganizes the CUDA instructions to avoid data dependency stalls and thus can hide some latency of memory access instructions. When the optimization is disabled with a $-O0$ flag, the table lookup for each byte is directly translated into CUDA instructions and the order would be the same.

In this AES GPU-based implementation, one GPU thread will perform AES encryption on one block of data. For a 32-block message, one warp of 32 threads can launch 32 encryptions in parallel. As the number of

blocks per message increases, the GPU throughput will increase.

Our server system is dedicated to performing AES encryptions, so we have configured the GPU to achieve high throughput. As the constant cache stores data from constant memory that can be shared by threads in a warp, for AES encryption we use this space to store round keys. All threads in a warp will access the same round key at the same time. This allows the constant cache to broadcast the value to all of the threads in the executing warp. Although the T-tables are also constant values, they will not benefit as much from using constant memory because each thread will generate different memory accesses and the constant cache would have to return them sequentially, wasting valuable resources. Therefore, we chose to place T-tables in the global memory, reducing the number of memory requests by leveraging the Coalescing units. Also, the T-table data can be shared across SMXs through the L2 caches and across warps in the same SMX through the L1 caches.

The L1 cache and the shared memory share the 64KB physical memory. Nvidia allows developers to determine this division on a per kernel basis. There are three available options: 16KB shared memory and 48KB L1 cache, 32KB shared memory and 32KB L1 cache, and 48KB shared memory and 16KB L1 cache. For our AES encryption kernel, the threads in a warp do not share any data, so the shared memory would not be used during the encryption. Thus, we want to minimize the size of shared memory and maximize the use of the L1 cache. The best configuration is to use 16KB shared memory and 48KB L1 cache.

During server initialization, the five T-tables are copied into the global memory, and the round keys are copied into constant memory. Constant data remains in the device memory until the application exits. During encryption, all memory requests to global memory access the L2 cache. However, by default, all global memory load/store operations bypass the L1 cache without being cached. We found that enabling the L1 cache can increase encryption performance. In this work, we will always configure the server program with L1 caching enabled.

4. CORRELATION TIMING ATTACK

We design a correlation timing attack that exploits the relationship present between the kernel execution time and the number of unique cache line requests generated during kernel execution. The attack will use one ciphertext byte and one *key byte guess* to compute the number of unique cache line requests that would be generated for the targeted ciphertext byte during table lookup. In attack, we run encrypt many messages and collect the timing information for each message (referred to as a *trace*, or data sample). We use the calculated number of cache line requests to correlate with timing samples. If we guessed the right key byte, we should expect to find a strong correlation between the timing and the correct number of unique cache line requests; otherwise, if we guessed the wrong key byte, the resulting correlation should be low.

In this section, we first explore the architecture leakage present on an Nvidia Kepler-family K40 device. To assess timing leakage vulnerabilities, we evaluate the success rates for correlation timing attacks using both *clean measurements* and *noisy measurements*. For the *clean measurements*, attackers are able to measure the warp execution time within a kernel, so the sources of inaccuracy are due to GPU internal hardware. When using *noisy measurements*, attackers are only able to measure when a message is received and returned by the server. In this case, the noise sources will also include processing on the server CPU, introducing some non-deterministic delay in our measurements. We consider the quality of timing data we collect, to better understand how noisy measurements can impact key recovery.

4.1 SIMT Architecture Leakage

With SIMT execution on a GPU, when a warp issues a load instruction, 32 memory requests by 32 threads are generated and sent to coalescing units (assuming all of the threads are active). These memory requests translate to unique cache line requests, and are merged with existing cache line requested in the MSHR. The time taken to serve all 32 memory address requests from a warp is proportional to number of unique cache line requests that are sent to the L1 cache controller due to both *memory issue serialization* and *writeback serialization*.

To determine if there is a linear relationship between an SIMT load instruction's execution time and the number of unique cache lines accessed, we develop a test kernel shown in Kernel 1. In this kernel, we measure the execution time for a warp of 32 threads to perform the load and store instructions in the kernel. Each thread is assigned an index from array *indices*, uses the index to load a 4-byte element from a big array A, and stores the element into the *result* variable. With SIMT execution of 32 threads, the array *indices* determine the total number of unique cache lines being referenced during the kernel execution, which is essentially sampling data array A. For example, if all the indexes are the same, Kernel 1 will only need one unique cache line. The array *indices* are created using Algorithm 2. In Algorithm 2, given a specified number of unique cache lines needed (e.g., 6), the array *indices* will generate the first six indexes with a stride of the cache line size, accessing six distinct cache lines, and the remaining 26 indexes will all be the same as the sixth one. With Algorithm 2, we can sweep the number of unique cache lines from 1 to 25 and generate corresponding *indices* arrays to use in Kernel 1.

With GPU's serialization memory handling, we expect the execution time to be linearly proportional to the number of unique cache line requests. In Figure. 3, we plot timing data for memory accesses while varying the number of cache line requests, under three strides, 32, 64, and 128 bytes. We can see that they are linearly

Kernel 1 The kernel to measure memory access time

```
index ← indices[tid]
time ← CLOCK()
result[tid] ← A[index]
time ← CLOCK() − time
```

Algorithm 2 Generating memory access indices that will result in a pre-set number of cache lines access for *Kernel 1*

```
numCacheLine ← userInput
indices ← []
curCacheLineIdx ← 0
for i = 1:25 do
    indices[i] ← curCacheLineIdx * stride
    if curCacheLineIdx < numCacheLine - 1 then
        curCacheLineIdx ← curCacheLineIdx + 1
    end if
end for
SHUFFLE(indices)
return indices
```

proportional, and that the slope of the linear lines indicates how much execution time is consumed for each unique cache access. A similar result was also reported in prior work [19]. Lines for stride size 64 and 128 bytes are exactly the same, implying that the cache line size is actually 64 bytes. From the Nvidia online literature [2], the cache line size of the L1 data cache of Kepler is 128 bytes. We confirmed this with Nvidia. What we also learned is that there are microarchitetural features in the L1 cache that are responsible for this behavior. As a result, we have elected to use 64 bytes as the cache line size in our attacks on the K40 device.

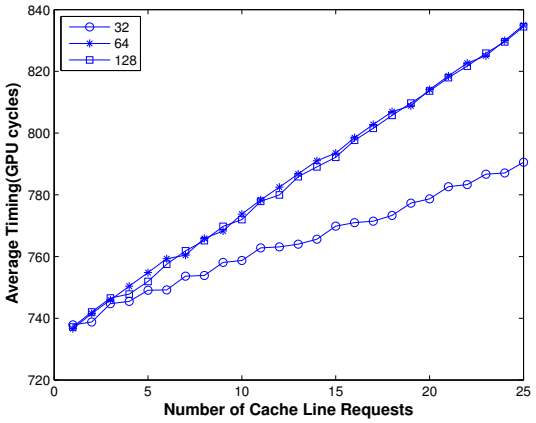

Figure 3: Nvidia GPU: Timing for 1 to 25 cache line requests, under stride 32, 64, and 128 bytes.

The Pearson Correlation value [20] of the execution time and the number of unique cache lines is found to be around 0.96. This strong correlation value suggests that the execution time can leak information about the array *indices* used in Kernel 1.

Not only does the Nvidia Kelper GPU exhibit this type of leakage, but AMD GPUs also show the same leakage. We performed the same timing analysis, running programs written in OpenCL on an AMD R9-290X GPU, with the stride set at 64 bytes (the AMD GPU L1 cache line size) and the result is shown in Figure. 4. We find its correlation value to be around 0.93

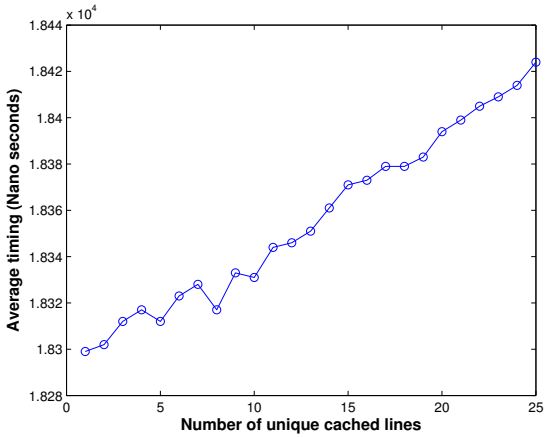

Figure 4: AMD GPU: Timing for 1 to 25 cache line requests.

Since SIMT and memory coalescing are crucial features for high performance GPU, this kind of correlation will persist in various GPUs, including the Nvidia and AMD GPUs that we have experimented with. Disabling either of them would significantly degrade the performance. We therefore expect to find this correlation on GPUs from other manufacturers as well.

Inspecting both Figure. 3 and Figure. 4, it is very clear that the execution time is directly proportional to the number of cache line requests on both families of GPUs. Given a fixed kernel, the key information (determines the number of cache line requests) can therefore be leaked by the execution time of the kernel. This keen observation inspires us to carry out a correlation timing attack on AES implementation running on the start-of-the-art GPUs.

4.2 AES Encryption Leakage

As shown in Figure. 1, the attacker and victim computers are connected via a network. This setup is the same as the one described by Bernstein. [12], except that the encryption is performed on the GPU. The goal of the attacker is to recover the 16-byte secret key that is used by the encryption server by using the known ciphertext and detailed timing collected. Depending on what timing data is collected, noise in this setup can be minimized if the execution time of kernels is measured. However, the measurement noise (i.e., inaccurracies) produced in a more practical setting will not inhibit the attack. As shown by Brumley et al., [21], the attacker can simply collect a larger number of traces and average out this noise.

Suppose that the attacker sends a 32-block message to the server, and the server launches a warp of 32 threads to encrypt the received data. After some time,

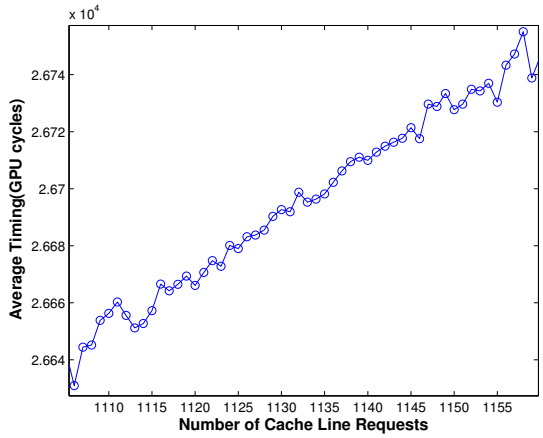

Figure 5: The average recorded time versus the total number cache line requests in a message encryption with one million samples

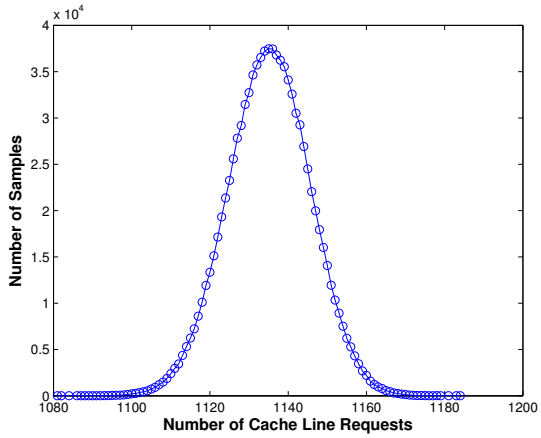

Figure 6: Sample counts versus the number cache line requests with one million samples.

the attacker receives the 32-block encrypted message, along with the timing information for the warp execution, which are stored as one timing trace (sample) as shown below:

$$\{c^1_{0-15}\ ,\ \ c^2_{0-15}\ ,\cdots,c^{32}_{0-15}\ ,T\}$$

There are ten rounds in AES encryption, and each round has 16 table lookups for each block of data. The index for each table lookup determines which cache line will be loaded. Thus, the entire encryption time depends on the indices for the 160 table lookups. We collected one million 32-block messages and associated timings, and recorded all indices used for the 160 table lookups during each block encryption. From all the indices used in the warp, we produce the number of unique cache line requests. We plot the average execution time associated with the number of unique cache line requests, shown in Figure. 5, as well as the sample counts that are used to calculate the average time in Figure. 6. Although the line in Figure. 5 does not appear as linear as the one shown in Figure. 3, it is clear that as the number of cache line requests increases, the average time also increases. The correlation between the number of cache line requests and the recorded execution time is 0.0596.

In a real attack, it is impossible to compute all of indices used during one encryption without knowing the entire 16-byte key due to the strong cryptographic confusion and diffusion functions. It will be computationally infeasible to enumerate the entire key space ($2^{128} \approx 3.4 \times 10^{38}$). However, in the last round, each lookup table index can be computed from one byte of the key and the corresponding byte of ciphertext, independently from other ciphertext bytes. Thus, we can examine how much leakage can be observed in one byte.

From Equation 1, we can write each byte of ciphertext as follows (byte positioning is ignored for simplicity):

$$c_j = T4[t_i] \oplus k_j \tag{2}$$

Using an inverse lookup table, we can find the i^{th} byte of the input state to the last round, t_i, if we know the true round key, k_j:

$$t_i = T4^{-1}[c_j \oplus k_j] \tag{3}$$

Given the GPU's SIMT execution model, for a 32-block message, we have 32 threads running simultaneously, and therefore:

$$t^1_i = T4^{-1}[c^1_j \oplus k_j]$$
$$t^2_i = T4^{-1}[c^2_j \oplus k_j]$$
$$\ldots$$
$$t^{32}_i = T4^{-1}[c^{32}_j \oplus k_j]$$

The values of table lookup indexes $t^1_i, t^2_i, ..., t^{32}_i$ determine the number of unique cache lines that will be generated. Since each element in the $T4$ table is 4 bytes and the size of a cache line is 64 bytes, there are 16 $T4$ table elements in one cache line (assuming the $T4$ table is aligned in memory). Therefore, the memory access requests can be turned into cache line requests by dropping the lowest 4 bits of $t^1_i, t^2_i, ..., t^{32}_i$, and so we have the following cache line requests:

$$\langle t^1_i \rangle = t^1_i >> 4$$
$$\langle t^2_i \rangle = t^2_i >> 4$$
$$\ldots$$
$$\langle t^{32}_i \rangle = t^{32}_i >> 4$$

The number of unique cache line is the number of unique values among $\langle t^1_i \rangle$, $\langle t^2_i \rangle$, $\cdots$, $\langle t^{32}_i \rangle$. This process of calculating the number of unique cache lines accessed from ciphertext bytes is implemented in Algorithm 3.

We generated one million 32-block messages and received one million encrypted messages, along with their associated timings. For each 32-block encrypted message, we used Algorithm 3 to calculate the number of unique cache line requests generated for $T4[t_3]$ table lookups, assuming we know the value of k_3. Figure. 7 shows the timing distribution over the number of cache line requests. We find the Pearson Correlation value to

Algorithm 3 Calculating the number of unique cache line requests in the last round for a given key byte guess.

$k_j \leftarrow guess$
$cache\_line\_cnt \leftarrow 0$
$holder \leftarrow [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$
%% comment: i means thread id
for i = 0:31 **do**
  $holder[T4^{-1}[cipher[i][j] \oplus k_j] >> 4] + +$
end for
for i = 0:15 **do**
  if holder[i] != 0 **then**
    $cache\_line\_cnt + +$
  end if
end for
return $cache\_line\_cnt$

be 0.0443. We also fit a linear line with a slope of 14 cycles and offset of 26503 cycles among the timing distribution, where the slope is taken as the signal for the leaking timing channel.

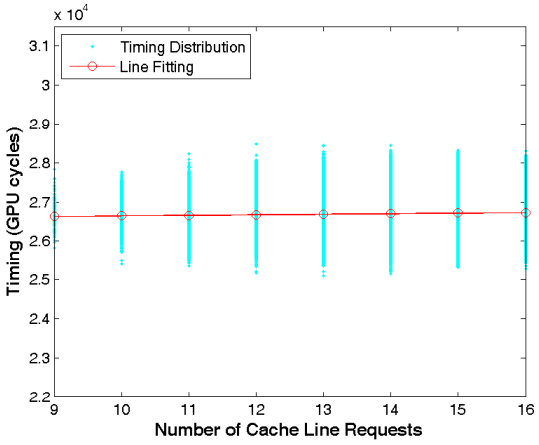

Figure 7: Timing distribution over the number of cache line requests, calculated for one million encrypted messages, using the true value of the 3^{rd} key byte.

Since we use only one table lookup out of all 160 table lookups for one block encryption, we should expect the Pearson Correlation to be bounded by the previously calculated correlation value for all 160 table lookups in Figure. 5. Although the correlation value gets small, it is still significantly higher than the correlation value when using the wrong value for the 3^{rd} key byte. If we assume the 3^{rd} key byte to be 0, we find its correlation to be 0.0012. That is 36.9 times lower than the correlation value calculated using the right key.

Although the correlation value is small, the linear relationship between the number of unique cache line requests and the encryption execution time suggests that the encryption time is leaking information about individual 9^{th} round cipher state. Since the 9^{th} round cipher state can be computed using ciphertext (known to the attacker) and key bytes, the encryption time ultimately is leaking individual key bytes.

4.3 Correlation Timing Attack on GPU AES Implementation

As we can see from Equation (1), each table lookup for each ciphertext byte is independent from another, and each key byte is used exclusively for its corresponding ciphertext byte. Thus, it allows us to attack one key byte at a time in a divide-and-conquer manner.

For each possible value of the key byte, we use Algorithm 3 to calculate the number of cache line requests for each 32-block message. Since timing is linearly proportional to the number of cache line requests in the last round, we computed the Pearson Correlation of the timing versus the number of cache line requests. When we guess the correct key byte, we have the correct number of unique cache line requests, and the resulting correlation should be the highest; on the other hand, if we guess the wrong key byte, the resulting correlation should be low. Therefore, the key byte guess with the highest correlation value among all possible values should be the correct key.

In this section, we test our Correlation Timing Attack method on the targeted Nvidia Kepler GPU. All of our experiments discussed in this section use one million traces, which can be collected within 30 minutes.

4.3.1 *Attack Using Clean Measurements*

In this experiment, we would like to demonstrate the feasibility of our attack first. Therefore, we minimize the noise by time-stamping when the kernel starts and ends as part of the AES kernel. This provides us with clean timing traces. The result of our attack is shown in Figure. 8. The correct value for each key byte is circled.

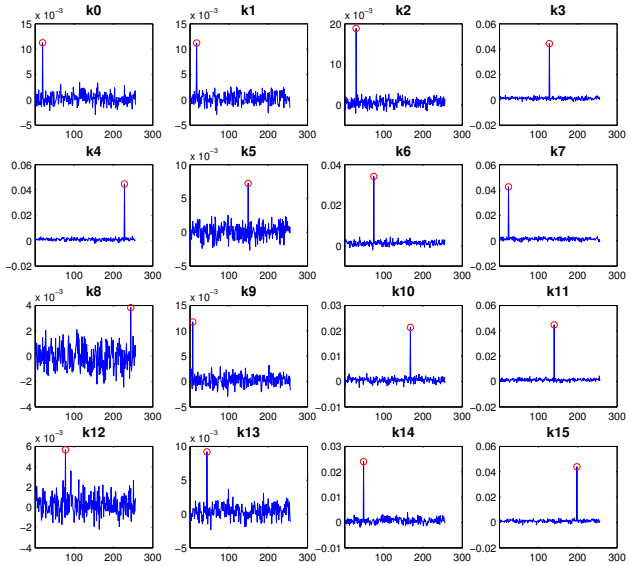

Figure 8: Correlation attack result under *clean measurements* for 32-block messages.

The correct key bytes stand out in the plots, when compared to the other 255 possible key values. This means we have successfully recovered all 16 bytes of

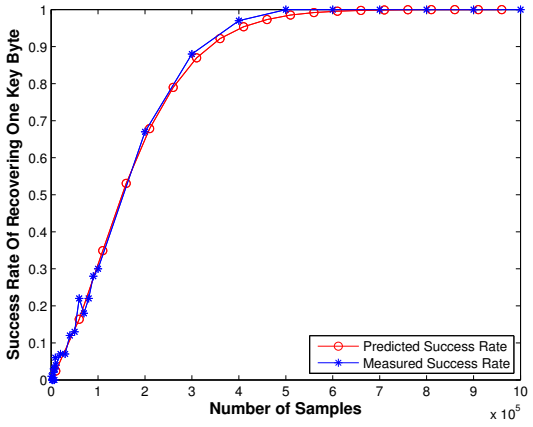

Figure 9: Success rate of k_5 using *clean measurements.*

the last round key. We also analyze the success rate of k_5 to see the number of traces needed for reaching different success probability (success rate). The result is shown in Figure. 9, which includes both the measured and predicted success rates. Each point on the measured success rate in the graph is the average from 100 timing attack trials using different timing traces. The predicted success rate is calculated using the methodology presented by Fei, et al. [22], in which the Signal to Noise Ratio (SNR) is obtained from real measurements (Figure. 7) and used to predict the success of recovering correct key value. The predicted success rate precisely predicts the one we measured. Since the predicted success rate tracks well with our measured results and computing the predicted success rate takes less than 30 minutes while computing the measured success rate using 100 trials takes around 1500 minutes, we will use the predicted success rate hereafter. From Figure. 9, both measured and predicted success rate reaches 50% using as few as 20,000 traces. With 70,000 traces, the predicted success rate converges to 1. Since the success rate is directly depending on the SNR value, we show SNR value for each byte in Table 1. As validated in Figure. 8, the correlation value for the correct key in each key byte varies. Thus, the SNR value for each key byte would also be different, as shown in Table 1, because there exists a linear relationship between the SNR and the correlation when the correlation value is small [22].

k_0	0.01	k_8	0.0034
k_1	0.01	k_9	0.0105
k_2	0.0168	k_{10}	0.0190
k_3	0.0395	k_{11}	0.0399
k_4	0.0399	k_{12}	0.0050
k_5	0.0064	k_{13}	0.0082
k_6	0.0305	k_{14}	0.0214
k_7	0.0379	k_{15}	0.0392

Table 1: The Signal to Noise Ratio (SNR) for each key byte.

Although the same lookup operation and same attack analysis are applied to each key byte, some key bytes, such as k_5, k_8, k_{12}, and k_{13}, have much smaller SNR values compared to others. This observation leads us to discover the effect of optimization during GPU compilation of the program on the timing attack.

To explore this issue deeper, we first examined how the server program is compiled for the GPU. We compiled our server program with the highest level optimization (-O3 by default in nvcc), which reorganizes some of the CUDA instructions in the kernel to avoid data dependency stalls. Before the table lookups for c_5, c_8, c_{12}, and c_{13}, other stalled load and store instructions may be congesting the GPU hardware resources, creating variable wait times for the table lookups for c_5, c_8, c_{12}, and c_{13}. We inspected the executable SASS (Shader ASSembly) code using the Nvidia dissembler. The code is shown in Listing 2, and includes parts of the last round operation. Line $2a38$ performs the table lookup for c_5, and the value loaded is not used until line $2a78$, when the GPU would stall if the requested data is not available at line $2a78$. Also, there are multiple load and stores instructions before line $2a38$, which may be congesting memory system and causing the duration of the load instruction for c_5 to be nondeterministic. Thus, we see very little correlation and a low SNR for those key bytes.

If we disable optimization during compilation, the CUDA instructions do not get reordered and each table lookup is stalled due to the data dependency and the timing actually is more deterministic or predictable. This is shown in Listing 1. Line $9ef0$ is the load instruction for c_5, and its requested data is needed immediately in the following instruction. The same happens to c_4 in line $9fa0$. Overall the non-optimized program runs much slower with a lot of stalls, 120,000 GPU cycles vs 27,000 GPU cycles for the optimized program.

Without the optimization, the loads of individual ciphertext bytes do not interfere with each other, and thus, we see a variance of 2354 GPU cycles square in timing data vs 128,000 GPU cycles square with optimization. The resulting execution timings of the load instructions is directly proportional to the number of unique cache lines accessed. Therefore, by performing the same attack with unoptimized server code, we would expect to have almost the same correlation value in each key byte, and the correlation is higher (around 0.06). The result is shown in Figure. 10.

If we disable optimization during compilation, the CUDA instructions do not get reordered and each table lookup is stalled due to the data dependency and the timing actually is more deterministic or predictable. This is shown in Listing 1. Line $9ef0$ is the load instruction for c_5, and its requested data is needed immediately in the following instruction. The same happens to c_4 in line $9fa0$. Overall the non-optimized program runs much slower with a lot of stalls, 120,000 GPU cycles vs 27,000 GPU cycles for the optimized program.

Without the optimization, the loads of individual ciphertext bytes do not interfere with each other, and

thus, we see a variance of 2354 GPU cycles square in timing data vs 128,000 GPU cycles square with optimization. The resulting execution timings of the load instructions is directly proportional to the number of unique cache lines accessed. Therefore, by performing the same attack with unoptimized server code, we would expect to have almost the same correlation value in each key byte, and the correlation is higher (around 0.06). The result is shown in Figure. 10.

Listing 1: Non-Optimized SASS Codes

```
...
/*9ef0*/ LD.E.64 R4, [R6];
/*9ef8*/ LOP.AND R4, R4, 0xff00;
  /*9f08*/ LOP.AND R5, R5, RZ;
  /*9f10*/ LOP.XOR R4, R22, R4;
  /*9f18*/ LOP.XOR R5, R23, R5;
  /*9f20*/ MOV32I R6, 0x0;
  /*9f28*/ MOV32I R7, 0x0;
  /*9f30*/ MOV R6, R6;
  /*9f38*/ MOV R7, R7;
  /*9f48*/ LOP.AND R8, R38, 0xff;
  /*9f50*/ LOP.AND R9, R39, RZ;
  /*9f58*/ SHF.L.U64 R3, R8, 0x3, R9;
  /*9f60*/ SHL R0, R8, 0x3;
  /*9f68*/ MOV R8, R0;
  /*9f70*/ MOV R9, R3;
  /*9f78*/ IADD R6.CC, R6, R8;
  /*9f88*/ IADD.X R7, R7, R9;
  /*9f90*/ MOV R8, R6;
  /*9f98*/ MOV R9, R7;
/*9fa0*/ LD.E.64 R6, [R8];
/*9fa8*/ LOP.AND R6, R6, 0xff;
...
```

Listing 2: Optimized SASS Codes

```
  /*2a00*/ ST.E.U8 [R6+0x3], R18;
  /*2a08*/ ST.E.U8 [R6+0x2], R14;
  /*2a10*/ IADD.X R13, RZ, c[0xe][0x24];
  /*2a18*/ IMAD.U32.U32 R16.CC, R17, R0, c[0xe][0x20];
  /*2a20*/ LD.E R11, [R10];
  /*2a28*/ LD.E R2, [R2];
  /*2a30*/ IMAD.U32.U32.HI.X R17, R17, R0, c[0xe][0x24];
/*2a28*/ LD.E R12, [R12];
  /*2a40*/ LD.E.64 R14, [R8+0x8];
  /*2a48*/ LD.E R16, [R16];
  /*2a50*/ LOP.AND R19, R22, 0xff;
  /*2a58*/ LOP.AND R22, R26, 0xff;
  /*2a60*/ LOP32I.AND R3, R11, 0xff000000;
  /*2a68*/ LOP32I.AND R2, R2, 0xff0000;
  /*2a70*/ SHR.U32 R11, R28, 0x15;
/*2a78*/ LOP.AND R10, R12, 0xff00;
...
```

Although we see much more consistency and higher correlation values in the attack result when the server code is not optimized, it is unlikely a high performance encryption engine would use unoptimized code. Therefore, we focus on using optimized server code to test our attack. While running the optimized server code, we are still able to recover all of the key bytes, and we have a better understanding of how optimization can begin to thwart timing attacks due to interference between loads.

4.3.2 *Attack Using Noisy Measurement*

In practice, it is more common for a server CPU to time-stamp the incoming and outgoing messages instead of within GPU kernels. With the same number of traces (one million) but different timing collection methods, we are able to recover 10 out of the 16 key bytes, as shown in Figure. 11. Changing from *clean measurement* to *noisy measurement*, we see the variance in timing data increases from 128 thousands to

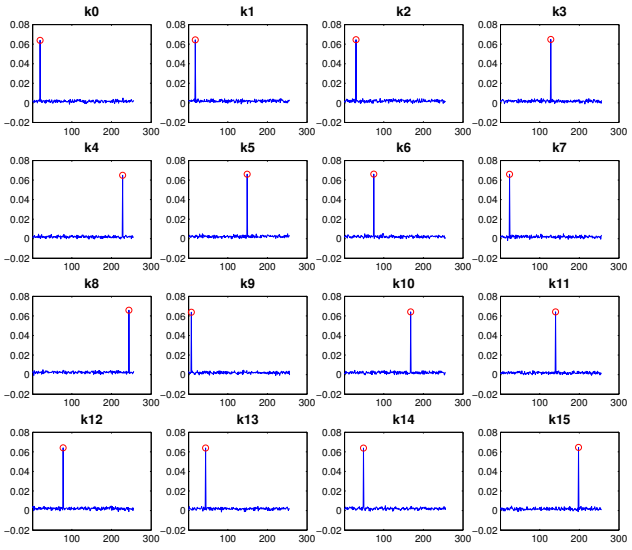

Figure 10: No optimization: Correlation attack result for 32-block messages.

5.8 millions GPU cycles, which means it introduce a lot of noise. The correlation exhibited in each key byte has been reduced by more than 3X, as compared to our previous results that assumed more accurate timing measurements.

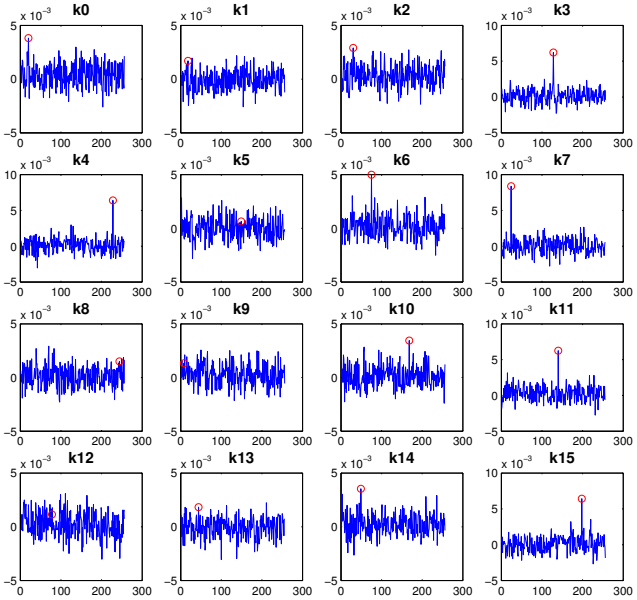

Figure 11: Correlation attack result with *noisy measurements* for 32-block messages.

Although adding noise in the timing hampers the attack slightly, it does not thwart the attack. With a large number of traces, we can achieve a 100% success rate at 3 millions traces. The attacker can use filtering to further clean up the timing information, and reduce the number of traces needed to achieve a 100% success rate. We applied a *Percentile Filter* as described by Crosby et al., [23] to our timing data. For data filtering, the

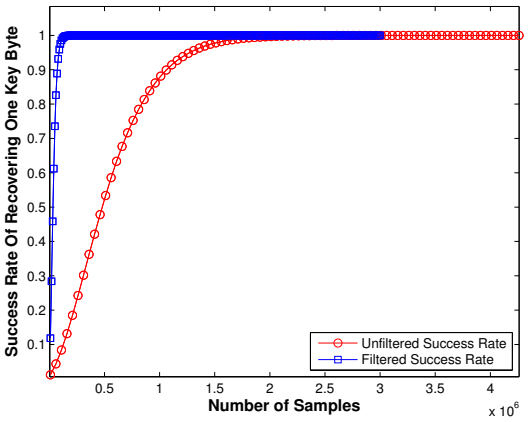

Figure 12: Predicted Success rate for 0^{th} key byte using filtered vs unfiltered data.

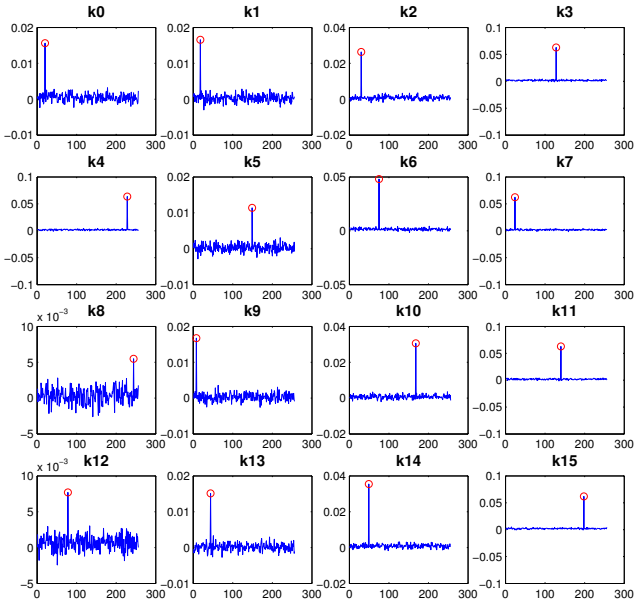

Figure 13: Correlation attack result using filtered timing information for 32-block messages.

attacker sends the same 32-block messages 100 times, so that she can obtain 100 timing samples, along with one 32-block encrypted message. Through experiments, we found that using the 40%percentile time among 100 timing traces produces the best attack result, as shown in Figure. 13.

By applying the simple noise reduction method, we are able to obtain even better results than obtained using *clean measurements*. Because even for *clean measurements*, the timing information also suffers from GPU-internal noise sources, such as uncertainty by the warp scheduler. By using noise filtering, most of noise sources are filtered out and result in much cleaner timing information. The success rate (Filtered Success Rate) shown in Figure. 12 converges to 1 at 40,000 traces, much earlier than the unfiltered scenario which requires 3 million traces. The simple filtering method significantly improves the attack effectiveness.

4.4 Attack on Highly Occupied GPU

Our experimental results suggest that GPU architectures with SIMT processing and a coalescing unit will produce a linear relationship between the number of unique cache line requests and the execution time. This relationship makes the GPU highly vulnerable to Correlation Timing Attacks. We can consider adding noise that would confuse attackers, but it does not fully thwart a timing attack. If we collect a large number of traces, attackers are still able to recover the secret information.

Our ability to extract the secret information from larger messages (e.g., 1024 blocks) is critical, because larger messages can better utilize the high throughput of the GPU. However, this might be unfavorable to the attacker. Unlike threads in a warp, threads in different warps are not synchronized, which implies some of warps may finish before others. Therefore, the longest warp execution time dominates the time measurement that the attacker is using. Although the attacker does not know which 32-block encrypted message is dominant, she can divide a 1024-block message into 32 32-block messages and treat them as 32 traces with the same time value. One of 32 traces, produced by the dominant warp, will have the true timing, and others might be wrong since the 31 other warps that produce the 31 others traces finished encryptions earlier than the dominant warp. Thus, the attacker can treat other 31 traces as noise to be added to the calculation. We collected one million traces using the filtering method discussed above. The results are shown in Figure. 14. Most key bytes are still recoverable, but the key bytes with weaker correlation, such as k_{12}, is completely buried. With more traces, k_{12} will also be recovered.

Since we treat the other 31 traces as noise during the calculation, we expect to see a success rate of close to 100% for 15 million traces, as shown in Figure. 15. Although increasing the number of blocks in each message may weaken the signal for each key byte, with larger number of traces, we can still recover all 16 key bytes.

4.5 Discussion

Moving from using *clean measurements* to *noisy measurements*, we see a lot of noise being included in our timing data, and consequently, the correlation value is suffering. In many real world situations, attackers would even not be able to get a time-stamp on the server. Thus, attackers would have to time-stamp their own packets as it being sent and received through the network. Such timing information can be much less accurate than the values in the *noisy measurements*. In the network setting, we observe the variance of the timing data to be $1.233e11$ CPU cycles square, comparing to $1.464e8$ in *noisy measurements* and $3.20e6$ in *clean measurements*. As discussed in [23], network noise can be filtered to make remote timing attack possible. At this moment, we are still exploring the attack in network setting.

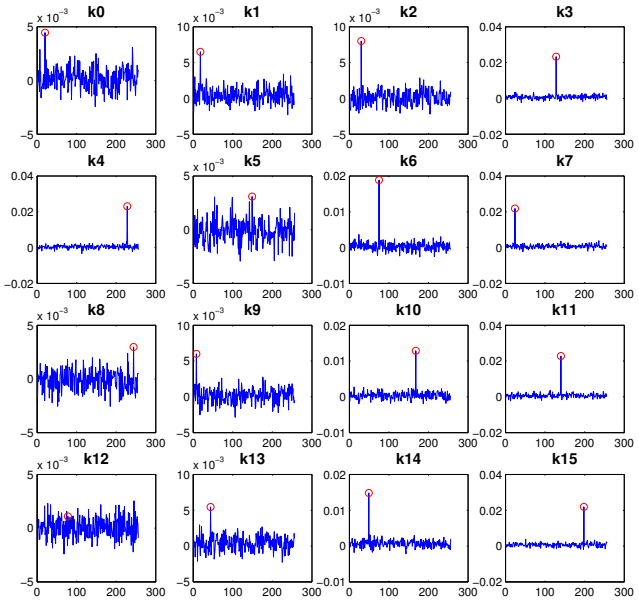

Figure 14: Correlation attack result using filtered timing information for 1024-block messages.

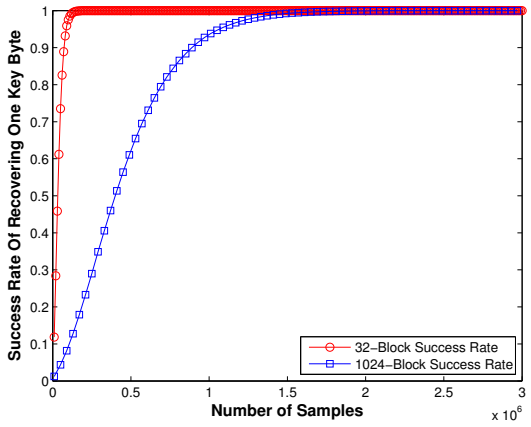

Figure 15: Predicted success rate for 0^{th} key byte using 1024-block data vs 32-block data.

5. COUNTERMEASURES

A large number of defense techniques have been proposed to avoid timing attacks on CPU platforms [24, 25, 26, 27, 28, 29, 30, 31]. Given the lack of study of side-channel vulnerabilities on GPU devices, there has been no prior work on GPU countermeasures. In this section, we discuss several potential mitigation methods.

Our attack exploits knowledge about the deterministic behavior of load instructions on the SIMT architecture. One method to prevent attacks is to table lookup operations in the AES implementation, as suggested by Osvik et al. [14]. We could also map the lookup tables to the GPU register file, since the register file is large enough hold a 256-byte Sbox table. This method can yield good performance, but it may limit the number of threads that can run on a SMX, since a SMX would not be able to launch more threads once its hardware resources are exhausted.

Another potential defense technique may be to introduce more noises. By increasing the noise, attacker would need to collect a larger number of samples in order to recover all of key bytes. Then the time and effort to collect data would also increase. The administrator can limit the number of samples that attacker can collect by changing the secret key on a frequent basis.

Our attack is possible because attackers are able to map a table lookup index to a cache line, so the attack would be infeasible if we randomize the mapping between a table lookup index to a cache line. The similar idea is also presented in [28], in which memory-to-cache mapping randomization is used on CPU platform. With this technique, given 32 table lookup indices, attackers would not be able to map them to cache lines, and would thus not be able to calculate the number of unique cache line requests. One possible implementation would be to randomize entries in the security data ($T4$) in the memory, and create a new index lookup table which maps an access index to the randomized index in the memory. Without knowing the mapping in the index lookup table, attackers would be not able to map an index to a cache line.

6. CONCLUSION

The execution time of a kernel is linearly proportional to the number of unique cache line requests generated during the kernel execution on modern GPU architectures. This property can be exploited to extract secret information such as encryption keys. In this paper, we exploit this property on a Nvidia GPU platform and successfully recover all 16 bytes of AES-128. Although we perform attacks on a Nvidia GPU platform, these attacks can be carried out on others GPU platforms, given that the SIMT feature and coalescing units commonly exist on GPUs.

In future work we plan to explore remediation techniques that thwart timing attacks on GPUs. We will also look at other kinds of side channels that may be present on a GPU device, including power analysis attacks and EM attacks.

Acknowledgment

This work was supported in part by National Science Foundation under grants SaTC-1314655 and MRI-1337854.

7. REFERENCES

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

[2] Nvidia, "Nvidia cuda toolkit v7.0 documentation," 2015.

[3] K. Iwai, T. Kurokawa, and N. Nisikawa, "Aes encryption implementation on cuda gpu and its analysis," in *Int. Con. on Networking & Computing*, pp. 209–214, 2010.

[4] S. Manavski *et al.*, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *IEEE Int. Conf. on Signal Processing & Communications*, pp. 65–68, 2007.

[5] A. E. Cohen and K. K. Parhi, "Gpu accelerated elliptic curve cryptography in gf (2 m)," in *IEEE Int. Midwest Symp. on Circuits & Systems*, pp. 57–60, 2010.

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

[7] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel aes algorithm for fast data encryption on gpu," in *Int. Conf. on Computer Engineering & Technology*, vol. 6, pp. V6–1, 2010.

[8] A. di Biagio, A. Barenghi, G. Agosta, and G. Pelosi, "Design of a parallel aes for graphics hardware using the CUDA framework," in *IEEE Int. Symp. on Parallel Distributed Processing*, pp. 1–8, May 2009.

[9] R. Di Pietro, F. Lombardi, and A. Villani, "CUDA leaks: information leakage in gpu architectures," *arXiv preprint arXiv:1305.7383*, 2013.

[10] M. J. Patterson, *Vulnerability analysis of GPU computing.* PhD thesis, Iowa State University, 2013.

[11] J. Danisevskis, M. Piekarska, and J.-P. Seifert, "Dark side of the shader: Mobile gpu-aided malware delivery," in *Information Security and Cryptology*, pp. 483–495, 2014.

[12] D. J. Bernstein, "Cache-timing attacks on aes."

[13] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems*, pp. 201–215, 2006.

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA*, pp. 1–20, 2006.

[15] Y. Yarom and K. E. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack.," *IACR Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

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

[17] C. Percival, "Cache missing for fun and profit," 2005.

[18] Nvidia, "Whitepaper nvidiaâĂŹs next generation cudatm compute architecture: Keplertm gk110," 2015.

[19] A. Lashgar, E. Salehi, and A. Baniasadi, "Understanding outstanding memory request handling resources in gpgpus," in *proceedings of The Sixth International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART'2015)*, 2015.

[20] K. Pearson, "Notes on regression and inheritance in the case of two parents," in *Proc. the Royal Society of London*, vol. 58, pp. 240–242, June 1895.

[21] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proc. Int. USENIX Security Symp.*, pp. 1–1, 2003.

[22] Y. Fei, A. A. Ding, J. Lao, and L. Zhang, "A statistics-based fundamental model for side-channel attack analysis.," *IACR Cryptology ePrint Archive*, vol. 2014, p. 152, 2014.

[23] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information & System Security*, vol. 12, no. 3, p. 17, 2009.

[24] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 494–505, 2007.

[25] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM Int. Symp. on Microarchitecture*, pp. 203–215, 2014.

[26] 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)*, vol. 8, no. 4, p. 35, 2012.

[27] D. Page, "Partitioned cache architecture as a side-channel defense mechanism.," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.

[28] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *IEEE/ACM Int. Symp. on Microarchitecture*, pp. 83–93, 2008.

[29] 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*, pp. 393–404, 2009.

[30] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," in *IEEE Int. Symp. on High Performance Computer Architecture*, pp. 225–236, 2014.

[31] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *IEEE/ACM Int. Symp. on Networks on Chip*, pp. 142–151, 2012.