Scatter-and-Gather Revisited: High-Performance Side-Channel-Resistant AES on GPUs

Zhen Lin
North Carolina State University
Raleigh, NC
zlin4@ncsu.edu

Utkarsh Mathur
North Carolina State University
Raleigh, NC
umathur@ncsu.edu

Huiyang Zhou
North Carolina State University
Raleigh, NC
hzhou@ncsu.edu

Abstract

Recent works have shown that there exist microarchitectural timing channels in contemporary GPUs, which make table-based cryptographic algorithms like AES vulnerable to side channel timing attacks. Also, table-based cryptographic algorithms have been known to be vulnerable to prime-and-probe attacks due to their key-dependent footprint in the data cache. Such analysis casts serious concerns on the feasibility of accelerating table-based cryptographic algorithms on GPUs. In this paper, we revisit the scatter-and-gather (SG) approach and make a case for using this approach to implement table-based cryptographic algorithms on GPUs to achieve both high performance and strong resistance to side channel attacks. Our results show that our SG-based AES achieves both high performance and strong resistance against all the known side channel attacks on these different generations of NVIDIA GPUs. We also reveal unexpected findings on a new timing channel in the L1 data cache (D-cache) on NVIDIA Maxwell and Pascal GPUs.

1 Introduction

Cryptography algorithms such as Advanced Encryption Standard (AES) are a critical foundation of information security. Although the AES algorithm remains mathematically secure to cryptoanalysis, various implementations of AES have been shown vulnerable to side channel attacks. In this paper, we focus on GPUs and revisit the scatter-and-gather approach for AES on GPUs so as to achieve both high performance and high resistance to existing side channel attacks including prime-and-probe attacks (aka access-driven attacks) and timing attacks.

Recent works have highlighted that the table-based AES implementation on GPUs, which is based on the AES code in OpenSSL [20], is vulnerable to timing channel attacks due to their memory coalescing logic and/or shared memory conflicts [11, 12]. The fundamental reason is that the correlation between the degrees of memory un-coalescing/bank conflicts and the execution time leaks the table index information through the timing channel. As the indices of the table lookups are dependent on the secret key, an attacker can derive the key by observing the timing differences when encrypting (or decrypting) pre-determined plaintexts (or ciphertexts). Besides timing information, it is also known that the table-based AES leaves key-dependent footprints in data caches, thereby becoming vulnerable to prime-and-probe attacks [21], aka access-driven cache attacks [14, 18], although such attacks require a more restrictive attack model that an attacker is able to set up and probe the cache used by the table-based AES.

To overcome the vulnerabilities of table-based AES, different AES implementations have been proposed such as bitsliced AES [23], in which a processor operates like a bit-level SIMD (single-instruction multiple-data) machine. Essentially, each bit in an n-bit processor works on one block (i.e., 128 bits) of the input. Therefore, each instruction in this implementation accomplishes one step for encrypting/decrypting n input blocks. Bitsliced AES eliminates all the tables and only utilizes logic operations, thereby immune to timing and prime-and-probe attacks. Unfortunately, it is vulnerable to differential power analysis or electromagnetic side channel attacks [2]. With n identical input blocks being fed to bitsliced AES, all the bits in a register will have the same value, either all 1s or all 0s, which are relatively easy to be distinguished using power or electromagnetic analysis, thereby leaking the intermediate results during encryption/decryption. This vulnerability can be more significant on GPUs than CPUs due to the vector-style registers in GPUs since the higher register width, the more distinguishable between the states of all 1s and all 0s.

In this paper, we aim to overcome all these vulnerabilities and retain high throughput close to table-based AES on contemporary GPUs. We first revisit the scatter-and-gather (SG) approach, which was proposed to implement Sbox-based AES on CPUs [3] and also used in RSA in OpenSSL (1.0.2f) [20]. In this work, we make the case for using SG in T-table-based AES on GPUs, the last round in particular. The main idea behind SG is to slice and re-organize

GPGPU-12, April 13, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6255-9/19/04…$15.00
https://doi.org/10.1145/3300053.3319415

the pre-computation tables such that key-dependent table lookups will not leak any timing or address pattern information. Also, the register values during encryption/decryption are randomized such that the intermediate results will not be leaked through power or electromagnetic analysis. We explore both Sbox based and T-table based AES and examine different SG granularities, which results in different performance and security trade-offs. We test the SG-based AES implementation on the D-cache in NVIDIA Kepler, Maxwell, Pascal, and Turing GPUs. Our results show that as long as we eliminate uncoalesced accesses, the D-cache in NVIDIA Kepler and Turing GPUs show constant latency and non-information-leaking footprint for all table lookups. The D-cache in Maxwell and Pascal GPUs, however, exhibits an undiscovered timing channel. Even when the accesses from all the threads in a warp fall into a single cache line, the load latency may vary. And this new timing channel can also be exploited to leak the key. Consequently, we choose to implement the SG-based AES using shared memory. Furthermore, as shared memory is statically allocated and cannot be replaced by a co-running kernel, it is inherently immune to prime-and-probe attacks and also the timing attacks relying on cache hit-miss patterns such as the collision attacks [5].

Our experimental results show that with the SG approach protecting both the first and last rounds of AES, we can achieve a throughput of 84 Gbps, 152 Gbps, 301 Gbps, and 261 Gbps on an NVIDIA Tesla K40, GTX 980, GTX 1080, and RTX 2080 GPUs, respectively. In comparison, the implementation using the special AES-NI instruction on an Intel Xeon CPU (E5-1607) achieves 27 Gbps throughput.

In this work, we make the following contributions. First, we make a case for SG to be used for table-based AES on GPUs to achieve both high performance and high resistance to all the known side channel attacks. Second, we identify a new timing channel in the L1 D-cache on NVIDIA Maxwell and Pascal GPUs. Third, we show that applying SG to T-table for the last round actually improves the performance rather than incurring overhead.

2 Background and Related Works

2.1 AES and Table-Based AES Implementation on GPUs

AES is a widely used symmetric-key block-cipher algorithm with the block size of 128 bits [7]. With different the key sizes of 128 bits, 192 bits, or 256 bits, it performs 10, 12, or 14 rounds of operations on the input data, respectively. In this work, we focus on AES encryption with 128-bit keys while the same discussion can be applied to the decryption process and other key sizes. In each round of AES cipher, there are multiple permutation, substitution, mixing, and logic operations involving the key. To speedup the process, pre-computed tables are used and each round can be implemented with a sequence of table lookups and bit-wise XOR operations with the round keys, which are expanded from the 128-bit key. In the T-table AES implementation from the OpenSSL library [20], each of the pre-computed tables has 256 entries and each entry has 32 bits. Therefore, each table has the size of 1kB. The output of one round is used as indices of the table lookups in the next round, except for the final round, whose output is the ciphertext.

To implement AES on GPUs, a standard approach is to leverage the data-level parallelism among the input data blocks [11, 12]. Each thread in a thread block (aka CTA) is responsible for generating one block (128 bits or 16 bytes) of ciphertext. There are two options to access the pre-computed T-tables. The first is to store the T-tables in the global memory and to leverage the L1 D-cache in each streaming multiprocessor (SM) for low-latency accesses. The second is to load the T-tables from global memory into on-chip shared memory within each SM. The trade-off is that the T-tables in shared memory are only accessible to the threads within the same thread block while the T-tables in the D-cache are accessible to all the threads running on the same SM, no matter which the thread block they belong to. As a result, if there are multiple thread blocks of the AES kernel running on an SM, there will be multiple redundant T-tables kept in shared memory if shared memory is used while only one set of the T-tables are loaded into the D-cache.

2.2 Side Channel Attacks against AES on GPUs

Although AES remains mathematically secure against cryptoanalysis, the table-based AES implementation is known to be vulnerable to side channel attacks. On CPUs, various cache attacks [15, 21] have been shown to be effective in deriving the keys. The reason can be explained through the last round of AES. Using a dedicated table T_4, which has 256 32-bit entries, the last round operation to produce the first byte of the ciphertext, c_0, can be described as: $c_0 = (T_4[t_3]\&0x000000ff) \wedge k_0$, where t_3 is the 4^{th} byte of the 16-byte state generated after 9^{th} round and k_0 is the first byte of the last round key. With c_0 known to an attacker, k_0 can be computed as $k_0 = c_0 \wedge (T_4[t_3]\&0x000000ff)$. The fundamental vulnerability is that if the index value to T_4 can be somehow determined, the key can be easily computed. To obtain such index information, prime-and-probe attacks set up the cache state with their own data and then let the last round of AES run. After that, by measuring the access latency of the originally cached data, an attacker can know which cache line(s) has been replaced, thereby stealing the index information. Although there could be more than one index resulting in the same replacement, with a few repeated prime-and-probes, the correct index can be revealed. AES on GPUs is vulnerable to the same prime-and-probe attacks as long as the D-cache is used for such table lookups.

Besides prime-and-probe attacks, new timing channels due to the memory coalescing logic and shared memory bank conflicts on GPUs have been recently identified [11, 12]. The GPU memory coalescing logic combines the accesses from all (or part) of the threads in a warp into as few numbers of cache lines as possible. The reason for the timing channel is that different numbers of shared memory conflicts or different degrees of memory un-coalescing lead to different load latency. In other words, the load latency leaks the information on the number of bank conflicts if shared memory is used for table lookups or the degree of memory un-coalescing (i.e., the number of cache line accessed) if the D-cache is used. An attacker can leverage the known ciphertext produced by all the 32 threads in a warp with a guessed key byte, e.g., k'_0, to perform an inverse table lookup, $t'_3 = T_4^{-1}[k'_0 \wedge c_0]$ for all the threads (different threads having different c_0). The attacker then uses t'_3 to perform a table lookup $T4[t'_3]$ and measure its latency. If the guessed key byte is correct, i.e., k'_0 being the same as k_0, t'_3 would be same as t_3 and the resulting bank conflicts or degree of un-coalesecing would also be the same. As different degrees of bank conflicts or un-coalescing might result in similar latency, multiple iterations of the attack with different plaintexts can be used to get high confidence. Pearson's correlation coefficient [22] can be used to extract linear correlations across different iterations. The correct guess of the key byte will have a high value of the Pearson's correlation coefficient as its latency information would be strongly correlated with the actual encryption latency.

To overcome the timing channel due to the memory coalescing logic, randomized memory coalescing [13] has been proposed, which adds noise to the measured latency and trades performance for resistance against the timing channel by always accessing a predetermined number of cache lines. As we will show in Section 5.2, there exists an undiscovered timing channel in the D-cache of some GPU models. Even if the timing channel due to memory coalescing is eliminated, the D-cache still may leak the key to an attacker.

Due to the vulnerability of table-based AES to both prime-and-probe and timing attacks, different implementations have been proposed. On CPUs, AES instructions, AES-NI [9], have been introduced to perform each round of computation on specialized hardware. A different implementation, bitsliced AES [23], was proposed to completely eliminate table lookups. However, it is shown that bitsliced AES is vulnerable to power or electromagnetic analysis [2] and is expected to be more so when implemented on GPUs as explained in Section 1. For table-based AES on GPUs, a prior work [16] also showed that power analysis may reveal substantial information.

Among the prior proposals on secure AES implementation, the one that is based on scatter and gather upon the Sbox [3] is considered secure on CPUs but suffers from low efficiency [3].

In this work, we propose to use the scatter and gather in the T-table based AES so as to achieve high resistance to all these above-mentioned timing and prime-and-probe side channel attacks on contemporary GPUs. In addition, we would like to retain the performance advantage of the table-based AES so as to achieve high throughput without special hardware support.

3 Scatter and Gather

3.1 Table Reorganization

As discussed in Section 2.2, the fundamental vulnerability for table-based AES is that the indices of the table lookups are key-dependent. Scatter-and-gather (SG) makes such dependency not observable by providing constant latency and non-information-leaking cache footprints.

Here, we use the last round of AES to motivate the idea. Considering the timing channel of memory coalescing logic on a GPU, the table lookup $T_4[t_3]$ for a warp of threads may result in different execution latency, which leaks the key. The table T_4 has 256 entries and each entry has 4 bytes. Different threads in a warp may produce different t_3 from their previous round. Therefore, with a cache line size of 128 bytes, this table lookup may span from 1 to 8 cache lines depending on the t_3 values from the threads in the warp. As a result, the latency of this table lookup is dependent on how many cache lines are actually accessed, which then leaks the information on the index t_3. However, after a careful look at the last round operation before XORing with the key byte, i.e., $(T_4[t_3]\&0x000000ff)$, we can see that only the least significant byte (LSB) of the 4-byte word is needed. In other words, although each thread loads a 4-byte word through this table lookup, only the LSB is needed, as highlighted in Figure 1. Then, an intriguing question would be: how about we extract and pack all the LSB of the 256 entries together into a new sub-table? We refer to this sub-table containing only the LSB of the words as T_{40} and use the data type of 'unsigned char' rather than 'unsigned int' for this sub-table. This way, the operation $(T_4[t_3]\&0x000000ff)$ would be reduced to a single table lookup ($T_{40}[t_3]$), and the logic operation extracting the LSB is no longer needed. More importantly, the footprint of this new table lookup becomes at most 256 bytes or up to 2 cache lines as the size of table T_{40} is 256 bytes, which are much reduced compared to the original T4 lookup. This reorganization leads to lower latency variation and less distinguishable cache footprints.

Recognizing the benefit of such a table reorganization, i.e., extracting and grouping the LSB of all the entries in a pre-computed table altogether, and then the next byte, and so on, we can make it more generic such that we can group an arbitrary slice of each table entry into a new sub-table. The granularity can be 1 bit, 2 bits, 4 bits, and 8 bits, and we refer to them as SG-1, SG-2, SG-4, and SG-8. As an example, SG-8 is the reorganization as shown in Figure 1.

Another way to look at the scatter-and-gather approach is to treat each table as a 2-dimension (2D) data structure. The

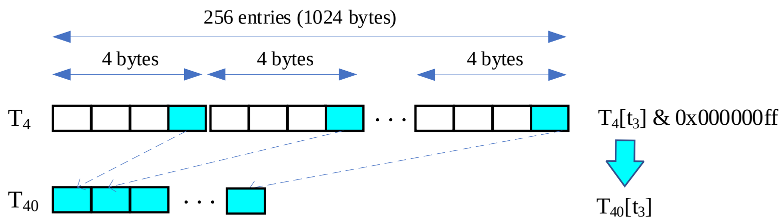


Figure 1. Reorganizing the table T_4 in the last round of AES. The LSB of each entry is put together as T_{40}, and the next byte as T_{41}, and so on.

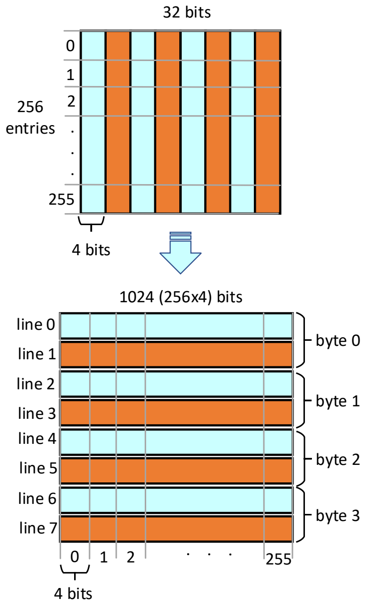


Figure 2. Scatter-and-Gather for 128-byte cache lines.

Y dimension is the number of entries in a table, e.g., 256. The X dimension is the number of bits/bytes in each entry, e.g., 32 bits. We scatter such a table into multiple columns and gather one column into a new sub-table, as shown in Figure 2. The width of a column is the SG granularity w. With SG, the original 2D data structure is reorganized into another 2D data structure. The new X dimension is $w \times 256$ and the new Y dimension is the number of sub-tables, which is $32/w$.

The choice of the SG granularity is to make sure that the resulting sub-table size is the same as a cache line or a cache sector, if a sectored cache structure [24] is employed. The reason is two-fold. First, with a sub-table residing in one cache line, all its table lookups are guaranteed to be coalesced, eliminating the timing channel due to the coalescing logic. Second, each lookup will bring in the entire sub-table as the fill granularity of the cache is a cache line (or a sector) and the index of a table lookup essentially becomes the offset within a cache line. Therefore, the footprint in the cache leaks no information on the table index at all. Given this SG granularity requirement and considering that the number of table entries is unchanged (i.e., 256), SG granularity can be computed as $w = cache\_line\_size/256$ for a particular cache design.

```
void SG4(uchar *OTAB, uchar *ITAB) {
    for (int i = 0; i < 8; i += 2) {
        for (int j = 0; j < 256; j += 2) {
            uchar b0, b1, lh, hh;
            b0 = ITAB[j*4 + i/2];
            b1 = ITAB[(j+1)*4 + i/2];
            lh = (b0 & 0x0f) | ((b1 & 0x0f) << 4);
            hh = ((b0 & 0xf0) >> 4) | (b1 & 0xf0);
            OTAB[i*128 + j/2] = lh;
            OTAB[(i+1)*128 + j/2] = hh;
        }   }    }
```

Figure 3. The code for scatter-and-gather with a granularity of 4 bits (SG-4).

With the cache line size of 128 bytes, the SG granularity of 4 bits ensures that an entire sub-table resides in one cache line as long as it is aligned at the cache line boundary. Figure 2 illustrates this case. Eight cache lines are required to store a T-table in its original format. After SG, the first sub-table stores bits 0-3 of all entries in the original table, and the second sub-table stores bits 4-7, etc.

Reorganization of the pre-computed tables can be viewed as another step in pre-computation and only needs to be done once for a particular SG granularity. Figure 3 shows the our proposed implementation to reorganize a T-table for the cache line size of 128 bytes, which implies the granularity of 4 bits. The input table, ITAB in Figure 3, is treated as a 256x4B matrix and the output table, OTAB in Figure 3, is a 8x128B matrix (or 8 sub-tables). The outer loop is used to traverse the 8 columns (two at a time) while the inner loop is used to traverse all the elements in every two columns. In each iteration, two vertically adjacent bytes are accessed. Then the lower halves of the two bytes are merged into a byte, *lh*, and the higher halves are merged into another byte, *hh*. At last, these two bytes are stored to the output table in the row-major manner such that one column in the input table is stored as one row in the output table.

The SG approach shown in Figure 3 can be easily adapted for various cache line sizes. In this paper, we implement SG-1, SG-2, SG-4 and SG-8 for 32B, 64B, 128B and 256B cache line/sector sizes, respectively.

3.2 Accessing the Reorganized Tables

With the SG mechanism, each table lookup instruction for all the threads in a warp can only access one cache line. With each entry of a sub-table contains w bits, each lookup

```
#define SG_4_LH(T, s, b) \
 (T[((b) << 8) + (s >> 1)   ] >> ((s & 0x1) << 2))
#define SG_4_HH(T, s, b) \
 (T[((b) << 8) + (s >> 1) + 128] >> ((s & 0x1) << 2))
#define FETCH_SG_4(T, s, b) \
 ((TS_4_LH(T, s, b) & 0x0f) | (TS_4_HH(T, s, b) << 4))
#ifdef SG_4
#define FETCH_SG(T, s, b) FETCH_SG_4(T, s, b)
#endif
```

Figure 4. The macros introduced for SG-4 table accesses.

only fetches w useful bits. As a result, multiple lookups or multiple sub-tables may be needed to return the required data. Specifically, fetching n bits requires n/w lookups. For example, as discussed before, with SG-8, the last-round table lookup ($T_4[t_3]$ & 0x000000ff) is simply converted to a single sub-table lookup, $T_{40}[t_3]$, as shown in Figure 1. With SG-4, in comparison, the same lookup ($T_4[t_3]$ & 0x000000ff) will be implemented with two sub-table lookups, with each returning 4 bits, and logic shift and XOR operations to merge the returned 4-bit values. Similarly, for a 32-bit table lookup $T_4[t_3]$, eight sub-table lookups are needed. The same index t_3 is used to get the 4-bit information from each sub-table. With each sub-table size as the cache line size, t_3 becomes the offset for the 4-bit value within a cache line.

In our proposed implementation, we define macros to facilitate the transformation from the original table lookups into the reorganized table lookups. Figure 4 shows the macros to transform a 1-byte access from the original tables to the SG-4 tables. In general, a 1-byte access to the original table is represented as $T[s]_b$, where s is an 8-bit index (as the tables have 256 entries) and b (ranging between 0 and 3) is the byte offset, denoting which byte of the 4-byte table entry to fetch. Using the macros, the access $T[s]_b$ is transformed to $SG\_4(ST, s, b)$, where TS is the reorganized table of T. As shown in Figure 4, $SG\_4(ST, s, b)$ is converted to two table lookups, TS_4_LH and TS_4_HH. As a result, two 128-byte cache lines are accessed to fetch one byte from the reorganized table. The reason is that with the SG granularity as 4 bits, each byte in the original table corresponds to 2 rows, as shown in Figure 2. The original byte offset b is used to determine the row index in the reorganized table or which sub-table(s) to be used. And the original table lookup index s is used as the offset of the desired 4 bits within a row/sub-table. Specifically, the higher 7 bits in s are used to locate the bytes within the 128-byte cache lines and the last bit in s is used to determine lower 4 bits or higher 4 bits to be used.

3.3 Integrating into the AES Algorithm

Figure 5 shows two code snippets from the AES implementation in OpenSSL v1.1.1 [20]. The statement in Line 1-7 shows a typical example of the operations in the first 9 rounds while the statement in Line 9-15 shows the last round. Note that in this implementation, there is no dedicated T_4 and $T_0 - T_3$ are reused instead. This implementation is functionally equivalent to the one with a dedicated T_4. We can see from Figure 5 that all the table lookups and bit-wise logic operations are performed at the 32-bit granularity even though only 8 bits are actually used in the last round table lookups.

```
1     // Code example of the first 9 rounds
2     t0 =
3         T0[(s0 >> 24)     ] ^
4         T1[(s1 >> 16) & 0xff] ^
5         T2[(s2 >> 8) & 0xff] ^
6         T3[(s3     ) & 0xff] ^
7         rk[4];
8     // Code example of the last round
9     s0 =
10        (T2[(t0 >> 24)     ] & 0xff000000) ^
11        (T3[(t1 >> 16) & 0xff] & 0x00ff0000) ^
12        (T0[(t2 >> 8) & 0xff] & 0x0000ff00) ^
13        (T1[(t3     ) & 0xff] & 0x000000ff) ^
14        rk[0];
```

Figure 5. Code snippets of the AES implementation in OpenSSL.

```
1   union u32_t {
2       uint u32;
3       uchar u8[4];};
4   #define LOAD_U8_M4(s, b) {            \
5       m0.u8[b] = FETCH_SG(T0, s0.u8[3], b); \
6       m1.u8[b] = FETCH_SG(T1, s1.u8[2], b); \
7       m2.u8[b] = FETCH_SG(T2, s2.u8[1], b); \
8       m3.u8[b] = FETCH_SG(T3, s3.u8[0], b); \}
9       // Code example of the first 9 rounds
10      u32_t m0, m1, m2, m3;
11      LOAD_U8_M4(s, 0); LOAD_U8_M4(s, 1);
12      LOAD_U8_M4(s, 2); LOAD_U8_M4(s, 3);
13      t0.u32 = m0.u32 ^ m1.u32 ^ m2.u32 ^ m3.u32 ^ rk[4];
14      // Code example of the last round
15      u32_t m;
16      m.u8[0] = FETCH_SG(T1, t3.u8[0], 0);
17      m.u8[1] = FETCH_SG(T0, t2.u8[1], 1);
18      m.u8[2] = FETCH_SG(T3, t1.u8[2], 2);
19      m.u8[3] = FETCH_SG(T2, t0.u8[3], 3);
20      s0.u32 = m.u32 ^ rk[0];
```

Figure 6. Code snippets of the AES implementation using the reorganized tables.

Using the $FETCH\_SG(T, s, b)$ macro defined in Figure 4, a byte can be accessed from the reorganized tables. One option to implement the SG-based AES is to perform bit-wise logic operations also at the 8-bit granularity. However, it requires separating a 32-bit operation into four 8-bit operations, which can significantly increase the number of ALU instructions. Instead, in our proposed implementation, as shown in Figure 6, we use a union data type $u32\_t$ to enable both 8-bit table lookups and 32-bit bit-wise logic operations. As shown in Line 14-15 in Figure 6, for the first 9 rounds, the table lookups are performed at the 8-bit granularity. Once all bits are in place, in Line 16, the bit-wise logic operations are performed at the 32-bit granularity. The implementation of last round is actually simpler than the original AES implementation because 8-bit table lookups eliminate the need of operations to extract the desired byte from the 32-bit table lookups.

	Tesla K40	GTX 980	GTX 1080	RTX 2080
Architecture	Kelper	Maxwell	Pascal	Turing
#SMs	15	16	20	46
L1 D$ size / SM	48KB/16KB	24KB	24KB	64KB/32KB
L1 D$ line size	128B	32B	32B	128B
Smem size / SM	48KB/16KB	96KB	96KB	64KB/32KB
Smem #Banks	32	32	32	32
Smem bank width	8B	4B	4B	4B

Table 1. Hardware specifications of the evaluated GPUs.

4 Methodology

In this paper, we evaluate 128-bit ECB mode AES encryption, which is ported from OpenSSL v1.1.1 to CUDA using the approach as described in Section 2.

We evaluate the SG-based AES on four Nvidia GPUs which span different architectural generations, including Tesla K40 (Kepler), GTX 980 (Maxwell), GTX 1080 (Pascal) and RTX 2080 (Turing). Table 1 lists hardware specifications of the GPUs. CUDA 10 is used for our experiments and the GPUs are hosted on a Red Hat 7.5 Linux machine. To make sure the global memory reads are always cached on L1, the compiler flag "-Xptxas -dlcm=ca" is used [19].

5 Security Analysis

5.1 Resistance to Timing Attacks

In this paper, to evaluate the resistance of our mechanism against timing attacks, we adopt the approach by Jiang et. al. [11] for attacking the GPU AES implementations. The attack relies on two sets of timing samples. One set is collected with the real key and the other is collected with the guessed keys. To reduce the timing noises, in each trial, we only launch one warp with 32 threads to encrypt 32 blocks of randomly generated plaintext. During each trial of the encryption with the real key, the attacker times the latency of every table lookup in the last round. There are 16 table lookups in the last round to produce 16 bytes (i.e., 1 block) of ciphertext. Then, another GPU kernel with 32 threads is launched to collect the timing with the guessed keys. Because each byte of the key can have 256 different values, it uses 256 values as each byte of the key to repeat the last round encryption. After N trials, the timing sample using the real key contains 16xN data points while the sample using the guessed keys contains 16x256xN data points. For each byte of the key, the Pearson correlation [22] is calculated between the sample with real key and the 256 samples with guessed keys. The byte of the key is successfully recovered if the guessed one with the highest correlation value matches the real one.

Using the correlation timing attack, we show how many bytes in the 128-bit key can be recovered for various AES implementations in Table 2 in 100,000 trials. Actually, because the timing information in our attack is collected in an ideal environment, the keys can be recovered within 100 trials if an implementation is vulnerable to timing attacks. As expected, the baseline T-table AES is vulnerable to the timing attacks on all GPUs. In the SG-8 implementation, each table lookup is constrained within 256 bytes, which is larger than the cache line size for all GPUs. Therefore, this implementation is also vulnerable. According to Table 1, the L1 data cache line size on Kepler and Turing GPUs is 128 bytes. Therefore, as expected, the SG-4, SG-2 and SG-1 implementations of AES become resistant to timing attacks on Tesla K40. The line size of Maxwell and Pascal GPUs are both 32 bytes. Therefore, it is reasonable that the SG-4 and SG-2 implementations are vulnerable. However, our results with our SG-1 AES are unexpected. As the SG-1 AES constrains each table lookup to 32 bytes, there should be no un-coalesced memory accesses and we expect that it would be resistant to timing attacks. However, our results indicate otherwise. Figure 7 shows the correlation coefficients of the first 4 bytes of the 128-bit key using the timing attack against the SG-1 AES on the GTX 1080 GPU. And the remaining 12 key bytes exhibit similar patterns. Clearly, the figure shows that the correct keys, which are circled, can be derived using the timing attack. The same attack also succeeds on the GTX 980 GPU against the SG-1 AES.

	Tesla K40	GTX 980	GTX 1080	RTX 2080
Base	16	16	16	16
SG-8	16	16	16	16
SG-4	0	16	16	0
SG-2	0	16	16	0
SG-1	0	16	16	0

Table 2. Number of bytes in the 128-bit key that can be recovered using the timing attack against various table-based AES using the D-cache in 100,000 trials.

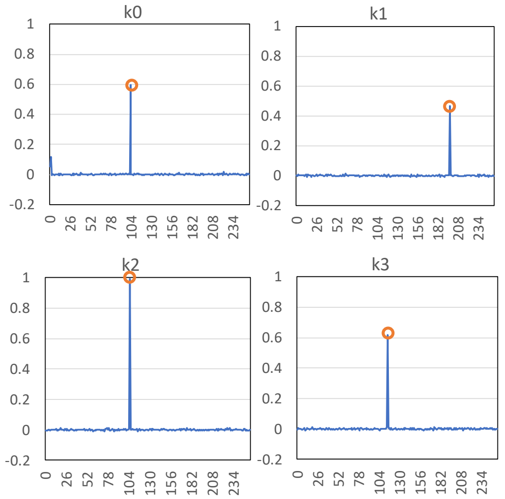


Figure 7. Pearson's correlation coefficients for guessing the first 4 bytes of the 128-bit key from AES with SG-1 tables on a GTX 1080 GPU.

5.2 An Undiscovered Timing Channel

In order to figure out the source that leaks the keys on Maxwell and Pascal GPUs, we use the micro-benchmark

```
// Host code: randomly init the indices.
srand(FIXED_SEED);
for (iter = 0; iter < N_ITERS; iter++) {
    for (i = 0; i < 32; i++)
        index[i] = rand() % LINE_SIZE;
    kernel<<<1, 32>>> (input, output, index, latency);
}
// Kernel code: time the access latency.
int idx = index[tid];
start = clock();
uchar a = input[idx]; // load one byte per thread
uchar b = a + 1; // make sure the load is done
latency = clock() - start;
```

Figure 8. The micro-benchmark for timing access latency of random access patterns.

as shown in Figure 8 to emulate the access patterns in our SG-based AES. In the host code, the index array is initialized with random numbers ranging from 0 to LINE_SIZE-1 and copied to the GPU. Then the kernel is launched with 32 threads. In the kernel code, each thread loads one byte from the input array using the values from the index array and we time the latency of the load operation. To make sure the results of the benchmark are repeatable, we use a fixed seed for the random generator in the host code to produce fixed random sequences in the index array for different invocations.

On the Tesla K40 and RTX 2080 GPUs, the latency is fixed if the indices from all the threads in the warp are within 0-127. However, on GTX 980 and GTX 1080 GPUs, even though the indices are generated within 0-31, the latency is not fixed for different random sequences. On the GTX 1080 GPU, the latency varies among 97, 98, 99, 100 and 101 cycles. Similarly, the latency varies among 98, 99, 100 and 101 cycles on the GTX 980 GPU. For a particular set of indices, however, the latency is fixed as long as the same set of indices repeats. However, we failed to observe a clear pattern between the random indices and the latency.

The exact reason for such variable latency is dependent on the D-cache design, which we don't have access to. However, even though the reason for this variable access latency is not clear, we are able to answer why our SG-1 AES is vulnerable to the timing attack on Maxwell and Pascal GPUs. In the timing attack, the access indices to the table are also randomized within the cache line size (i.e., from 0 to 31). In different trials, different plaintexts lead to different sets of access indices. In the guessing runs, when the guessed key matches the correct key, the index sequence for a warp would exactly match the indices in the encryption run, which leads to the same access latency. Using an enough number of trials, the guessed key byte exhibits the highest correlation with the correct key byte. Note that this side channel is different from the one due to the memory coalescing logic that has been studied in previous works [11, 13]. Even when the accesses from all the threads in a warp are coalesced to a cache line, the access latency is not constant and the keys can still be leaked on Maxwell and Pascal GPUs. Therefore, using global memory for table lookups, the SG-based AES is not secure on Pascal and Maxwell GPUs against timing attacks.

	Tesla K40	GTX 980	GTX 1080	RTX 2080
Base	16	16	16	16
SG-8	0	15	14	16
SG-4	0	0	0	0
SG-2	0	0	0	0
SG-1	0	0	0	0

Table 3. Number of bytes in the 128-bit key that can be recovered using the timing attack against various table-based AES using shared memory in 100,000 trials.

5.3 Resistance to Prime-and-Probe Attacks

As discussed in Section 3, our design decision for the SG granularity is to ensure that each row / sub-table fits in one cache line. Not only does it eliminate un-coalesced accesses, but also it converts the original table lookup index into offset with the cache line. And this provides the foundation for our defense against prime-and-probe attacks. Taking SG-4 as an example for a cache with the cache line size of 128 bytes. As shown in Figure 5, the table lookups in AES have two granularities, 32 bits for table lookups in rounds 1-9 and 8 bits in the last round. A table lookup in the last round is converted to two sub-table lookups (or cache line accesses) as shown in Figure 2 and Figure 4. Therefore, with a prime-and-probe attack, an attacker can know which byte (and either the higher 4 bits or lower 4 bits of the byte) is accessed. But such information carries no secrets as the AES source code already indicates which byte to be accessed in a table. The table index, which carries the key dependent information, is not leaked as it is converted the offset within the cache line. The code transformation is shown in Figure 5 also explicitly shows that the table index *s* is used as the offset within each sub-table.

A table lookup in round 1-9 fetches 32-bit information. In this case, with SG-4, all the 8 sub-tables / rows will be accessed with each providing 4-bit information as can been seen from Figure 2.2. In other words, the entire pre-computed table is accessed and no table index information can be leaked. Therefore, we conclude that SG-based AES is effective against prime-and-probe cache attacks.

6 SG with Shared Memory

So far, we focus on SG-based AES using global memory, meaning that the table lookups are through the D-caches. In Section 5, we observe that our approach is not completely secure due to an unexpected timing channel on Maxwell and Pascal GPUs. In this section, we discuss the security implications for SG-based AES using shared memory, i.e, when the pre-computed tables are stored in shared memory.

As introduced in Section 2, a previous work by Jiang et al. [12] shows that table-based AES using shared memory is vulnerable to timing attacks. The reason is that the shared memory access latency is dependent on the number of bank conflicts, which leaks the index information of a table lookup. As long as we can avoid bank conflicts, a table lookup in shared memory will leak no secret information. We achieve this using the SG approach discussed in Section 3.

In this paper, we refer to "a row" in shared memory as an aggregation of storage units with the same row address in all the banks. Similar to SG-based AES using the L1 D-cache, the key idea here is to reorganize the table so that each table lookup to the shared memory is constrained to a single row. The mechanisms of SG-based AES with shared memory is identical to what is presented for D-cache in Section 3. The only difference is that the tables need to be loaded from the global memory at the beginning of each thread block before the cipher process.

6.1 Resistance to Timing Attacks

To verify whether the accesses have constant latency when accessing the addresses within a single row in shared memory, we resort to the same micro-benchmark as shown in Figure 8. The only change is that the input array is allocated on shared memory and the LINE_WIDTH is replaced with ROW_SIZE. Our test results show that on the Tesla K40 GPU, which has a 256-byte row size, the access latency is constant at 74 cycles. On GTX 980 and GTX 1080 GPUs, the row size is 128 bytes and the latency is constant at 52 cycles, independent upon the different random indices. The access latency on the RTX2080 GPU is constant at 56 cycles.

We perform the timing attack as described in Section 5.1 against SG-based AES with shared memory. Table 3 shows how many bytes (out of 16) can be recovered with 100,000 trials for various table-based AES implementations. The entire key can be recovered for the baseline AES within only 100 trials on any GPU in our experiment. The SG-8 AES, which constrains one access within 256 bytes, is resistant to the timing attack on the Tesla K40 GPU because its row size is 256 bytes. For GTX 980, GTX 1080 and RTX 2080 GPUs, however, the SG-8 version leaks 15, 14 and 16 bytes, respectively, since the row size is 128 bytes on these GPUs. In contrast, the SG-4 AES is resistant to the timing attacks on GTX 980, GTX 1080 and RTX 2080 GPUs because one access can only span a region of 128 bytes, which is the row size of the shared memory on these GPUs. The SG-2 and SG-1 AES implementations are resistant to the timing attack on all the GPUs since their table lookup is smaller than the row sizes of shared memory.

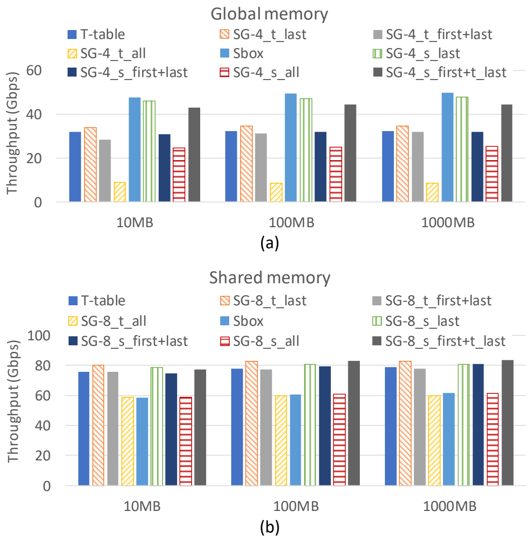
6.2 Resistance to Prime-and-Probe Attacks

On GPUs, shared memory space is allocated when a thread block is dispatched. Then, the pre-computed tables are fetched into shared memory before the cipher process starts. Such shared memory data, i.e., the tables, will remain there throughout the lifetime of the thread block and cannot be replaced or evicted. Therefore, AES implementations using shared memory for its tables is inherently immune to prime-and-probe attacks. Furthermore, since all table lookups are guaranteed to retrieve their data from shared memory, these accesses are all ‘hits’. Therefore, the attacks based on hit/miss patterns, such as collision attacks, are not effective.

7 Performance Results

In this section, we evaluate the throughput of SG-based AES encryption on different GPUs with various plaintext sizes. Four AES implementations are evaluated in every test. ‘Baseline’ is the AES version that we ported from OpenSSL. As discussed in Section 5 and 6, it is vulnerable to the timing attack with table lookups in either global memory or shared memory. The label ‘SG-w’ denotes the SG-based AES with the w-bit granularity. In the ‘SG-w_last’ approach, we only apply the reorganized table to the last round, which is the most vulnerable round and used as the target of previous GPU timing attacks [11, 12]. In ‘SG-w_first+last’, we apply the reorganized table to both the first and last rounds while the original tables are used for the remaining rounds. As pointed out in prior works [4, 6, 8], the first and last rounds are more susceptible to side channel attacks than inner rounds as the input/output of the inner rounds are processed with the roundkeys while the plaintext/ciphertext is the input/output of the first/last round. The prior work [6] also suggested that it is probably sufficient to protect both the first and last round because of this reason. In ‘SG-w_all’, the reorganized tables are used for all rounds, providing the highest resistance against side channel attacks. We include the SG approach used for the T-tables and Sbox, denoted by SG-w_t and SG-w_s, respectively. The tradeoff is that the Sbox approach accesses a smaller lookup table than T-tables but requires more computations. For the last round, as accessing the reorganized T-table is more efficient than the reorganized Sbox, we also explore the option of using the reorganized Sbox for the first round and the reorganized T-table for the last round, denoted as ’SG-w_s_first+t_last’. All these approaches are implemented with tables residing in either global memory or shared memory. In the shared memory implementations, to reduce the number of shared memory loading times, every thread iteratively encrypts the plaintext blocks until all data blocks are finished. For each implementation on different GPUs, the grid size and thread block size parameters are tuned to achieve the best performance. The source code our implementations are available online [1] for result reproducibility and further security analysis.

Figure 9(a) and (b) show the throughput of various AES approaches on the Tesla K40 GPU with the tables residing in global memory and shared memory, respectively. According to Table 2 and 3, SG-4 is used for global memory while SG-8 is used for shared memory to ensure their resistance



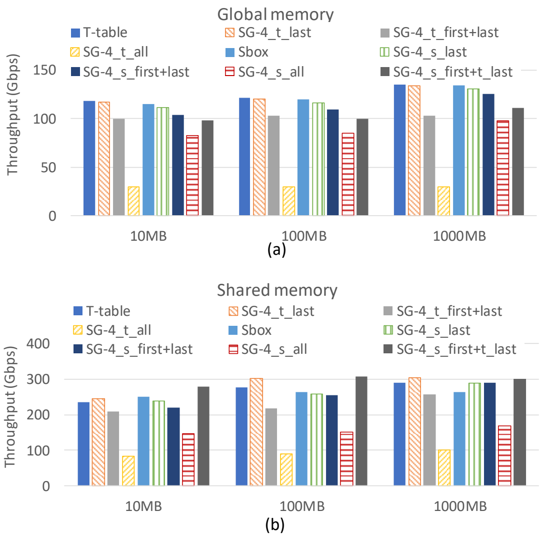
(a)

(b)

Figure 9. Throughput of SG-based AES on a Tesla K40 GPU.

to side channel attacks. On Kepler GPUs, the same on-chip memory is used for the L1 D-cache and shared memory and the capacity can be configured to prefer the L1 cache or shared memory. In our experiment, we configure the preference to the L1 D-cache when running the global memory implementation while the preference is set to shared memory when running the shared memory version. Comparing to the baseline, the performance is actually improved by applying the reorganized table only to the last round. This is because the table accesses in the last round are performed at byte granularity, as shown in Figure 5, and the accesses to the reorganized T-tables are guaranteed to be coalesced or bank-conflict free. However, in the first 9 rounds, SG requires multiple byte accesses and bit-wise logic operations to merge them into 32 bits, while the same 32-bit information can be accessed with just one load instruction using the original T-tables. Therefore, the performance is significantly affected. In order to mitigate the performance loss and also remain the resistance to all the known attacks, we choose to apply SG to the first and last rounds. The throughput of 'SG_s_first+t_last' approach is 45 Gbps and 84 Gbps with global memory and shared memory, respectively.

Figure 10 shows the performance of SG-based AES on the GTX 1080 GPU. According to Table 3, SG-4 is used for shared memory implementation. Although none of the global memory implementations on GTX 1080 is secure, we show the performance of SG-4 for performance comparison purposes. The shared memory implementations achieve about 2x throughput than the global memory one due to lower latency and higher bandwidth. According to previous works [10, 17], on Pascal GPUs, the shared memory latency is 3.4x lower than the L1 D-cache and the bandwidth is 4x higher. The 'SG-4_s_first+t_last' approach achieves 301 Gbps using shared memory.



(a)

(b)

Figure 10. Throughput of SG-based AES on a GTX 1080 GPU.

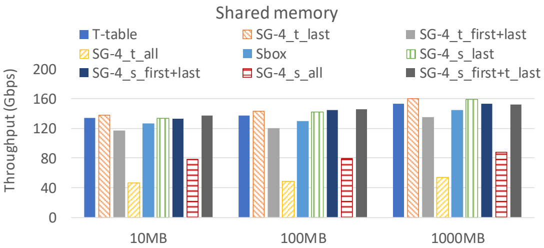
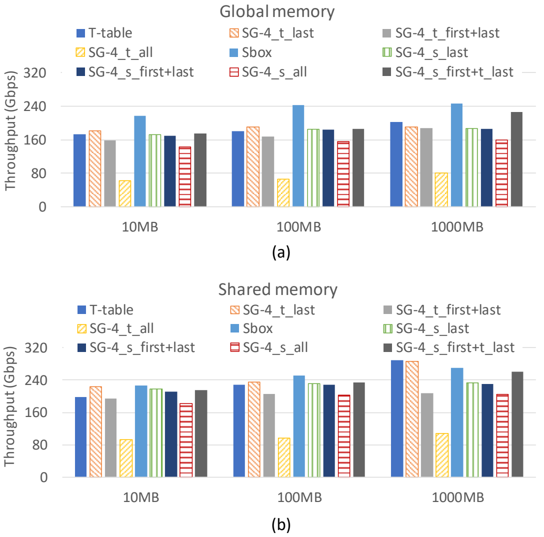


Figure 11. Throughput of SG-based AES on a GTX 980 GPU.



(a)

(b)

Figure 12. Throughput of SG-based AES on a RTX 2080 GPU.

Similar to the GTX 1080 GPU, as shown in Figure 11, we use SG-4 with shared memory for the GTX 980 GPU. The throughput achieved by the 'SG-4_s_first+t_last' approach is 152 Gbps.

According to Table 2 and 3, on the RTX 2080 GPU, SG-4 AES using both global memory and shared memory is resistant to timing attacks. Similar to the Kepler architecture, the L1 cache and shared memory share the same on-chip memory. According to the CUDA Programming Guide [19], the capacity preference is automatically selected by the driver to achieve the highest performance. The 'SG-4_s_first+t_last' approach achieves 227 Gbps using global memory and 261 Gbps using shared memory.

8 Conclusions

In this paper, we make a case for scatter-and-gather to implement table-based cryptography algorithms on GPUs. Recent works show that the current table-based AES is vulnerable to side channel attacks on GPUs due to the timing channels in shared memory or memory coalescing logic. The scatter-and-gather based approach reorganizes the tables such that the key-dependent table index information becomes not observable anymore. Not only is this transformation expected to be effective against timing attacks, but also is effective against prime-and-probe style attacks. Our study on four NVIDIA GPU generations also reveals an undiscovered timing on the D-cache of both Maxwell and Pascal GPUs. As a result, we conclude that our SG-based AES, using the reorganized Sbox for the first round and the reorganized T-table for the last round, with shared memory achieves the highest resistance to side channel attacks and also offers the best performance, 84 Gbps, 152 Gbps, 301 Gbps, and 261 Gbps on a Tesla K40, GTX 980, GTX 1080, and RTX 2080 GPU, respectively, which is 105%, 99%, 104%, and 90% of the throughput of the original (unsecure) T-table based AES from OpenSSL on these GPUs, respectively.

Acknowledgements

We would like to thank Lars Nyland and Jin Wang from NVIDIA for their insightful discussions. We thank the anonymous reviewers for their valuable comments. This work is supported by NSF grants CCF-1618509, CCF-1717550.

References

[1] [n. d.]. https://github.com/zhenlin36/scatter_gather_aes_cuda

[2] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. 2015. DPA, Bitslicing and Masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, Sep 2015, Proceedings*. 599–619.

[3] Johannes Blömer and Volker Krummel. 2007. Analysis of Countermeasures Against Access Driven Cache Attacks on AES. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*. 96–109.

[4] Joseph Bonneau. 2006. Robust Final-Round Cache-Trace Attacks Against AES. *IACR Cryptology ePrint Archive* 2006 (01 2006), 374.

[5] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*. 201–215.

[6] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. jean-pierre.seifert@intel.com 13192 received 13 Feb 2006.

[7] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer. https://doi.org/10.1007/978-3-662-04722-4

[8] Qian Guo, Vincent Grosso, and FranÃğois-Xavier Standaert. 2018. Modeling Soft Analytical Side-Channel Attacks from a Coding Theory Viewpoint. Cryptology ePrint Archive, Report 2018/498. https://eprint.iacr.org/2018/498.

[9] Intel. 2010. *Intel Advanced Encryption Standard (AES) New Instructions Set* .

[10] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826

[11] Z. H. Jiang, Y. Fei, and D. Kaeli. 2016. A complete key recovery timing attack on a GPU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 394–405.

[12] 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, USA, 167–172. http://doi.acm.org/10.1145/3060403.3060462

[13] G. Kadam, D. Zhang, and A. Jog. 2018. RCoal: Mitigating GPU Timing Attack via Subwarp-Based Randomized Coalescing Techniques. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 156–167.

[14] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. 2013. Architecting against Software Cache-Based Side-Channel Attacks. *IEEE Trans. Computers* 62, 7 (2013), 1276–1288.

[15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA*. 605–622.

[16] 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 Computer Society, Washington, DC, 281–288.

[17] X. Mei and X. Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan 2017), 72–86.

[18] Michael Neve and Jean-Pierre Seifert. 2006. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*. 147–162.

[19] NVIDIA. 2018. *CUDA C PROGRAMMING GUIDE*.

[20] OpenSSL. [n. d.]. *url: https://www.openssl.org/*.

[21] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. 1–20.

[22] Karl Pearson. 1895. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London* 58 (1895), 240–242. http://www.jstor.org/stable/115794

[23] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. 2006. Bitslice Implementation of AES. In *Cryptology and Network Security, 5th International Conference, CANS 2006, Suzhou, China, December 8-10, 2006, Proceedings*. 203–212.

[24] André Seznec. 1994. Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio. In *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*. 384–393.