BCoal: Bucketing-based Memory Coalescing for Efficient and Secure GPUs

Gurunath Kadam
William & Mary
Williamsburg, VA
gakadam@email.wm.edu

Danfeng Zhang
Pennsylvania State University
University Park, PA
zhang@cse.psu.edu

Adwait Jog
William & Mary
Williamsburg, VA
ajog@wm.edu

***Abstract*—Graphics Processing Units (GPUs) are becoming a de facto choice for accelerating applications from a wide range of domains ranging from graphics to high-performance computing. As a result, it is getting increasingly desirable to improve the cooperation between traditional CPUs and accelerators such as GPUs. However, given the growing security concerns in the CPU space, closer integration of GPUs has further expanded the attack surface. For example, several side-channel attacks have shown that sensitive information can be leaked from the CPU end. In the same vein, several side-channel attacks are also now being developed in the GPU world. Overall, it is challenging to keep emerging CPU-GPU heterogeneous systems secure while maintaining their performance and energy efficiency.**

In this paper, we focus on developing an efficient defense mechanism for a type of correlation timing attack on GPUs. Such an attack has been shown to recover AES private keys by exploiting the relationship between the number of coalesced memory accesses and total execution time. Prior state-of-the-art defense mechanisms use inefficient randomized coalescing techniques to defend against such GPU attacks and require turning-off bandwidth conserving techniques such as caches and miss-status holding registers (MSHRs) to ensure security. To address these limitations, we propose BCoal – a new bucketing-based coalescing mechanism. BCoal significantly reduces the information leakage by always issuing pre-determined numbers of coalesced accesses (called buckets). With the help of a detailed application-level analysis, BCoal determines the bucket sizes and pads, if necessary, the number of real accesses with additional (padded) accesses to meet the bucket sizes ensuring the security against the correlation timing attack. Furthermore, BCoal generates the padded accesses such that the security is ensured even in the presence of MSHRs and caches. In effect, BCoal significantly improves GPU security at a modest performance loss.

***Index Terms*—GPUs, Hardware Security, Coalescing**

I. Introduction

Graphics Processing Units (GPUs) provide orders of magnitude higher throughput compared to CPUs thanks to a large number of computational units attached with high bandwidth memory. GPUs have traditionally accelerated a wide-range of arguably security insensitive applications ranging from gaming to high-performance computing. However, many applications that benefit from GPUs nowadays process or contain security/privacy-sensitive information. For example, DNA and financial computing applications that heavily process private data are taking advantage of GPUs [1], [2]. The deep learning community has significantly benefited from the computational power of GPUs but now is also concerned about the privacy of their models and vendors; they are interested in protecting them from motivated attackers [3], [4]. Cryptographic and other computations that handle sensitive data are also known to achieve significant performance benefits from GPUs [5]–[11].

With the growing need for secure GPU computation, it is important to protect GPUs from a variety of possible side-channel attacks. For example, several attacks (especially, cache-based side-channel attacks [12]–[18]) on the CPU side have exploited the fact that critical information can be leaked if it affects the latency (or total execution time). In the same vein, new correlation timing attacks and covert channels [6], [19]–[21] are being exposed in GPUs – a recent attack [6] showed that AES private keys can be recovered by exploiting the correlation between the number of coalesced accesses and execution time. Specifically, an attacker exploits the relationship between the private keys and the number of coalesced accesses to reveal the entire private key by performing off-line correlation analysis with the help of recorded execution time and encrypted (cipher) text information.[1]

Kadam et al. [5] presented the first work to address the aforementioned correlation timing attack. They showed that by randomizing the logic of coalescing unit (RCoal), additional accesses can be generated such that the correlation between the baseline (real) accesses and the execution time is reduced. Consequently, the attacker finds it hard to recover the private keys. However, we find that RCoal has two major drawbacks. First, the performance loss for security gain is very high due to the randomization of coalescing logic, especially for large plain texts. Second, RCoal provides sub-optimal security in the presence of other memory bandwidth conserving mechanisms such as miss-status holding registers (MSHRs) and caches. As we further demonstrate in Section III, the additional duplicate accesses generated during randomization are merged back in MSHRs to render RCoal ineffective. Therefore, RCoal turned-off caches and MSHRs for security reasons, leading to even more significant performance overheads.

To efficiently address the limitations of RCoal, we propose a new bucketing-based coalescing technique – BCoal. It always generates the number of coalesced accesses equal to one of the pre-determined values (known as buckets), irrespective of program secrets. This implies BCoal would generate additional

[1] More details on the attack are provided in Section II.

memory accesses (if necessary) along with the real accesses to match the bucket requirements. As the number of accesses is always equal to the pre-determined values, the variance in the number of accesses drops. As a result, BCoal reduces the correlation to mitigate the timing attack.

To reduce the performance overhead of additional accesses, we select optimal bucket features by analyzing the application-level coalescing profile. The goal of profiling is to select the bucket features such that overall fewer additional accesses are generated. Further, we observe that the generation of additional accesses is non-trivial because we need to ensure that they affect the execution time at the same rate as the real accesses, otherwise their effect on the execution time can be filtered out (i.e., noise can be filtered out from signal). To address this issue, we generate *unique* additional accesses to the same memory space as that of the real accesses. We find that this helps in reducing the disparity between caching/merging probabilities of real accesses and additional accesses, thereby making their individual effects on execution time also similar. Consequently, our bucketing-based coalescing technique provides security even in the presence of MSHRs and caches.

To the best of our knowledge, this is the first work that proposes a bucketing-based coalescing technique for GPUs to achieve better security compared to the state-of-the-art scheme while incurring low overhead. In summary, this paper makes the following contributions:

• We perform a detailed analysis to show that the state-of-the-art defense schemes against the coalescing-based correlation timing attack are inefficient. They incur a significant performance and data movement overhead as they work only when the bandwidth conserving hardware such as caches and MSHRs are not employed.

• We propose a new bucketing-based coalescing mechanism (BCoal) that always issues pre-determined numbers (chosen from a small set, called buckets) of coalesced accesses by padding additional accesses to the real accesses, if necessary.

• Our analysis shows that the generation of padded accesses is non-trivial and the effect of MSHRs and caches should be considered to ensure security. BCoal implements a *homogeneous* padding mechanism to ensure that the real and padded accesses affect the execution time *similarly* even in the presence of MSHRs and caches. Therefore, an attacker fails to separate the timing effect of padded accesses thereby improving the security.

• Our theoretical and experimental analysis shows that BCoal significantly improves the security (i.e., drops the correlation by up to 100%) at a modest performance overhead ranging from 5% to 15%. We also evaluate BCoal across a large set of GPGPU applications and show that coalescing with three equally-spaced buckets provides an excellent performance-security trade-off that can be leveraged to secure the GPUs.

II. Background

This section briefly introduces: a) the baseline GPU architecture, b) bandwidth conserving mechanisms, c) the AES encryption on GPU, and d) the baseline correlation timing attack and the state-of-the-art defense mechanism against it.

A. Basics of GPU Architecture

We consider a baseline GPU architecture with multiple cores, known as streaming multiprocessors (SMs) in NVIDIA terminology. The SMs are connected to memory partitions via an interconnect as shown in Figure 1. GPUs achieve high throughput by executing a large number of threads concurrently. To facilitate this, GPUs are supported by a large register file (for fast context switching across threads) and high bandwidth memories (for fast data access to a large number of concurrent threads). Each SM executes the threads assigned to it at the granularity of a *warp*, which is essentially a collection of (usually 32) individual threads that execute a single instruction on the processing elements (PEs) of the SM in a lock-step. The warps hide long memory latencies to improve the utilization/throughput of the SM via executing in a pipelined and multiplexed manner. Throughout the paper, we evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [22]. Table I provides details of the simulated architecture.

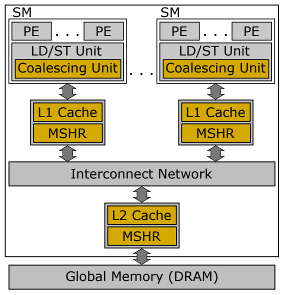


Fig. 1: Overview of Baseline GPU Architecture.

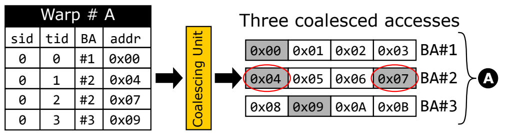
TABLE I: Key configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file, 15 SMs
L1 Caches / Core	16KB 4-way L1 data cache, 2KB 4-way I-cache 128B cache block size
L2 Caches	16-way 256 KB/memory channel (1536 KB in total), 128B cache block size
Features	Inter-warp merging enabled
Memory Model	6 GDDR5 Memory Controllers, FR-FCFS scheduling 16 DRAM-banks, 924 MHz memory clock
Interconnect	1400MHz interconnect clock

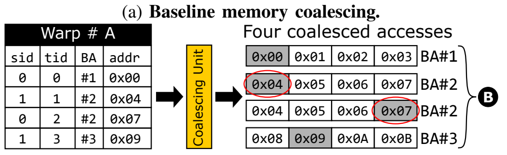
B. Bandwidth Conserving Mechanisms

Memory bandwidth is one of the most performance-critical shared resources in GPUs [23], [24]. GPUs adopt several memory bandwidth optimization techniques, such as memory access coalescing, caching and merging to reduce the number of accesses to the global memory. In this sub-section, we provide a brief overview of these optimizations.

Access Coalescing. In GPUs, threads within a warp execute the instructions in lockstep. For a global memory load instruction, all 32 threads within a warp execute 32 load instructions. The coalescing unit in the LD/ST unit merges multiple memory requests from different threads of the same warp (*intra-warp coalescing*) into as few cache line-sized coalesced memory accesses as possible. The intra-warp coalescing happens at the sub-warp granularity, where the coalescing unit of the SM determines the coalesced accesses of the warp by examining a group of threads belonging to the same sub-warp. If the threads of a sub-warp access data within a contiguous memory block, their requests are coalesced together to reduce memory bandwidth consumption. The size and number of sub-warps are typically fixed and remain the same throughout the application execution. However, to achieve security, the coalescing mechanisms can be randomized (RCoal [5]) so that the coalesced accesses are no longer predictable to the attacker.



(a) **Baseline memory coalescing.**



(b) **Randomized memory coalescing.**

Fig. 2: Memory access coalescing in GPUs.

Figure 2 illustrates the coalescing in baseline GPU and previously proposed randomized coalescing techniques. Assume a single warp with four threads. The per-thread addresses and the requested block addresses (BA) are shown with corresponding thread-ids (tid) and sub-warp id (sid). In the baseline GPU, we assume a single sub-warp (sid = 0 for all threads) and hence all threads participate together in the coalescing. Since the requests from tid 1 and 2 map to the same cache block, only three accesses are generated (A) to conserve the memory bandwidth. With randomized coalescing, the threads are randomly assigned to subwarps and hence lead to unpredictable effects on coalescing. In Figure 2(b), we observe that four accesses are generated (B) due to different sub-warp ids assigned to the random groups of thread. More details on the randomized coalescing techniques are discussed in Section II-D.

Caching. GPUs further conserve the memory bandwidth by exploiting the temporal and spatial locality in memory accesses across and within warps with the help of hardware caches. Current GPUs employ two levels of caches, L1-cache (shared by the warps executing on the same SM) and L2-cache (shared by the warps executing on different SMs).

Access Merging. The coalesced memory accesses from a warp are sent to the L1-cache. Upon cache misses, the memory accesses are logged in the miss-status holding registers (MSHRs). Multiple cache-missed coalesced accesses to the same cache block from different warps on the same SM are merged (*inter-warp merging*) in MSHRs. Note that as independent loads from the same warp can be issued to improve memory-level parallelism, MSHRs also help in merging redundant accesses from the same warp (*intra-warp merging*) if they are issued at different times. Another source of inter-warp merging is via MSHRs at L2-cache, where the redundant L2-cache misses (across different SMs) can be merged together.

C. AES Encryption

To demonstrate the GPU timing attack exploiting the vulnerability due to memory coalescing, we consider the widely used symmetric-key algorithm, Advanced Encryption Standard (AES) [25]–[29] with a key length of 128 bits, to encrypt the plaintext. AES-128 algorithm consists of 10 rounds, each with a 16-bytes round key generated from the encryption key. We focus on the last round of the AES, which is shown to be the most vulnerable to side-channel attacks [6]. The last round involves a table (for the S-box table T_4) look-up operation followed by bitwise XOR operation with the last round key.

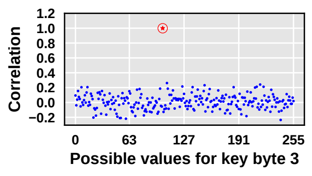
Our AES implementation on GPU is from Jiang et al. [6], [11], which was used in the original attack [6] and a known defense [5]. We used the same implementation for a fair comparison. The AES implementation on GPU involves dividing the plaintext across multiple parallel threads to achieve high throughput. Each thread encrypts a line of the plaintext independent of other threads. Therefore, a warp consisting of 32 threads can perform 32 different encryptions concurrently. In general, the line to thread mapping is sequential and deterministic. If the size of the plaintext exceeds 32 lines, then it is divided sequentially among several warps. For example, a plaintext with 1024 lines will employ 32 warps each executing 32 lines of the plaintext. To ensure a stronger baseline for comparison, the AES implementation used in this paper performs random mapping of threads to the warps (known as input blinding) to gain additional security [5].

D. Baseline Attack and Defense Mechanism

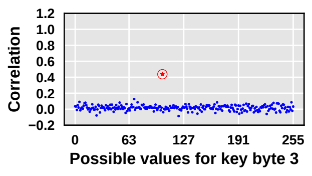
Baseline Attack. In this work, we use the same attack model as designed by Jiang et al. [6]. It assumes that the attacker can send a large number of plaintexts to a remote GPU-based AES [25]–[29] encryption server and collect the ciphertext. The attacker also records the total execution time required to complete each encryption. The attack was also shown to be very effective in noisy environments [6].

Given that the GPU coalescing procedure is deterministic [30] and the last round of AES is invertible [6], the attacker can calculate the number of coalesced accesses with the help of ciphertext and a last round key guess. As the number of coalesced accesses is correlated with the execution time in the baseline system [6], the key guess that leads to the best correlation across a large number of encryptions is determined to be the correct key. This attack further assumes that the round tables are kept in GPU DRAM, which can be cached in L1/L2 caches based on the access patterns. For brevity, we skip the algorithmic details of the attack and refer readers to prior works [5], [6]. Also, the rest of the paper assumes a stronger attacker with the capability of accessing last round execution time as compared to the realistic attack, which is weaker due to the noise in the total execution time. Consequently, we assume the goal of the attacker is to correctly guess the last round AES encryption key [5], which can divulge all other round keys by reverting the fixed AES key generation schedule.

Figure 3 shows the scatter plots for the baseline correlation attack for the single-warp (plaintext with 32 lines) and multi-warp (plaintext with 64 lines) cases. Each scatter plot shows the correlation values for all 256 possible values for the 3^{rd} key byte of the last round. Each point on the scatter plot



(a) **Plaintext with 32 lines.**



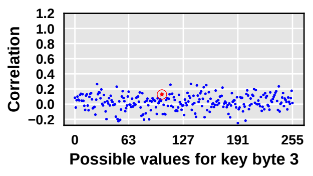
(b) **Plaintext with 64 lines.**

Fig. 3: Baseline Attack.

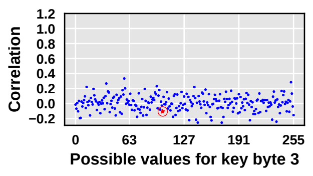
corresponds to a correlation value between the number of coalesced accesses (on a per-warp basis) calculated by the attacker and the execution time of the last round of AES-128. In the multi-warp case, the maximum value of the number of coalesced accesses across all warps is used as the warp that generates the most number of coalesced accesses has shown to dominate the total execution time [6]. From Figure 3, we observe that the correlation value is the highest (highlighted in red and encircled) for the correct value of the 3rd key byte among all other guess values for the single- as well as the multi-warp case. Therefore, the correct value of the key byte 3 is recoverable. We observe this trend for all last round key bytes indicating successful recovery.

Baseline Defense. Kadam et al. [5] presented a series of randomized coalescing (RCoal) mechanisms to defend against the correlation timing attacks. They showed that randomizing the number of subwarps, the sizes of subwarps, and the thread elements of the subwarp can improve the GPU security, however at the cost of performance loss and increased data movement between SMs and memory. Based on these three parameters, three RCoal mechanisms were proposed: fixed-sized subwarp (FSS), random-sized subwarp (RSS), and random-threaded subwarp (RTS). They showed that the best performance-security trade-off can be achieved with an RCoal mechanism (RSS+RTS+4), which uses the number of subwarps to be 4, the sizes of warps are chosen based on a skewed distribution, and the thread elements are chosen randomly based on a uniform distribution. In rest of the paper, we denote this best of the RCoal scheme as RCoal(4). Note that if the number of subwarps is equal to the number of threads in a warp then it is equivalent to coalescing being disabled as all threads independently participate in the coalescing procedure. For example, with a warp size of 32, choosing the number of subwarps to be 32 is equivalent to disabling the coalescing. We denote this as RCoal(32). RCoal(32) was shown to be the most secure design as the number of coalesced access is always constant at 32 [5]. Due to security concerns, RCoal disabled caches and MSHRs (refer to Section III-B for more details).

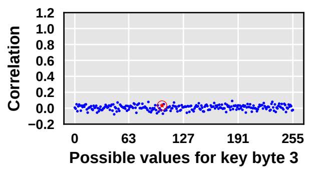
Figure 4 shows the scatter plots for RCoal(32) (the most secure mechanism) and RCoal(4) (best of RCoal) using plaintext with 32 and 64 lines. In contrast to the baseline attack, for RCoal(32) and RCoal(4), the correlation between the number of coalesced accesses and execution time with the correct key (highlighted in red and encircled) dropped significantly. Consequently, this point is no more distinguishable among the other correlation points ensuring successful defense against the attack. We observe this trend for all last round key bytes.



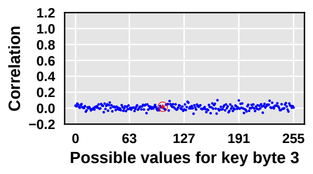
(a) **RCoal(4) with Plaintext (32 lines).**



(b) **RCoal(32) with Plaintext (32 lines).**



(c) **RCoal(4) with Plaintext (64 lines).**



(d) **RCoal(32) with Plaintext (64 lines).**

Fig. 4: Effect of different RCoal coalescing schemes on the recovery of one of the last round key byte (shown in red circle). In RCoal, the caches and MSHRs are disabled for security reasons (refer Section III-B).

III. Motivation and Analysis

Although RCoal helps in improving the GPU security significantly, it also incurs a very high performance and data movement overhead. To substantiate the overhead of RCoal, Figure 5 shows the total execution time and number of DRAM accesses for two scenarios: a) RCoal(32) – the most secure design, and b) RCoal(4) – the best of RCoal. These results are shown for three different sizes of plaintexts (32, 64, and 1024) and are normalized to the baseline GPU. We observe that the overhead of RCoal(32) is very high – more than 27× increase in the number of DRAM accesses leading to over 9.4× increase in the execution time. Furthermore, the performance degradation increases rapidly with the size of plaintexts. The same trend is visible for RCoal(4) as well.

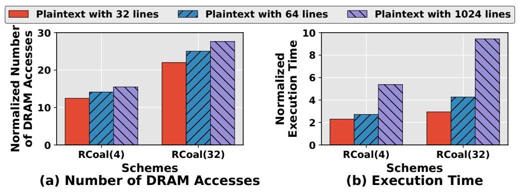
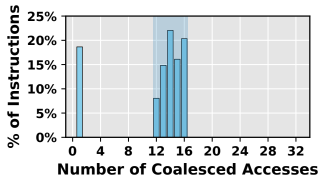
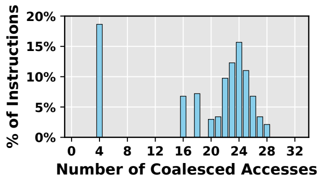


Fig. 5: Illustrating the overhead of RCoal defense scheme for different sizes of plaintext. The results are normalized to a baseline GPU with MSHRs and caches.

A. *Performance Overhead Analysis of RCoal*

There are two major reasons behind the large performance and data movement overhead. First, RCoal introduces sub-optimal and randomized coalescing that causes additional memory traffic. To understand this, we analyze the number of coalesced accesses generated in three different architecture options: baseline, RCoal(4), and RCoal(32). For these three options, Figure 6 shows the number of coalesced accesses with respect to the percentage of load instructions in the AES CUDA implementation. We observe a bimodal distribution in


(a) **Baseline.**


(b) **RCoal(4).**

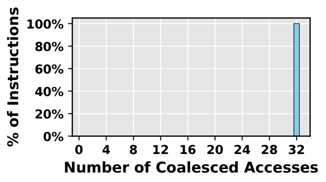

(c) **RCoal(32).**

Fig. 6: Histogram of the number of coalesced accesses generated across a warp for 1000 plaintext samples each with 32 lines.

the baseline scenario (Figure 6(a)): the first peak occurs when only one coalesced cache line access is generated for roughly 20% instructions and the second peak occur between 12-16 coalesced cache line accesses for the remaining instructions. The first peak is observed due to the loads for the round keys and the second peak is due to the table lookup operations. With RCoal(32) (Figure 6(c)), the coalescing unit performs worst to always generate 32 coalesced accesses for all load instructions. As noted before, this is similar to the coalescing being disabled. Although it is the most secure option, the average number of coalesced accesses and the overall number of DRAM accesses increase significantly (Figure 5). In RCoal(4) (Figure 6(b)), we observe that the second peak has shifted to the right compared to Figure 5(a) due the obfuscation of the coalescing mechanism that generates additional memory traffic. Overall, RCoal(4) and RCoal (32) generate additional memory traffic and incur performance penalties to reduce the correlation between the number of baseline coalesced accesses and the execution time. Importantly, RCoal ignores the application properties, especially the baseline coalescing profile to optimally generate the traffic while reducing the correlation.

Second, due to the security reasons, RCoal schemes were only shown to work in the absence of other bandwidth optimization techniques, such as caches and MSHRs. The absence of MSHRs and caches has a substantial impact on the performance and data movement, and is well-documented in GPU literature [22], [23], [31]. The combined effect of sub-optimal coalescing, and absence of MSHRs and caches leads to a sharp increase in the number of DRAM accesses resulting in high performance degradation.

B. Effect of MSHRs and Caches on Security with RCoal

Effect of MSHRs. In the presence of MSHRs, RCoal scheme becomes vulnerable to the correlation timing attacks. RCoal randomizes the access coalescing and generates redundant accesses to the same block addresses to reduce the correlation between the execution time and the number of baseline coalesced accesses. The MSHRs render RCoal scheme ineffective by merging the redundant accesses to the same block addresses leading to similar correlation as in the case of baseline GPU. The effect of MSHRs on RCoal scheme is prominent for the table lookup instructions experiencing a high cache-miss rate as the corresponding accesses are likely served through MSHRs leading to predictable access merging. This is especially true for the initial table lookup instructions of the last round because T_4 table elements are less likely cached. Figure 7 shows this merging-back phenomenon using the example from Figure 2. RCoal(4) generated 4 accesses Ⓐ, including one redundant access. However, MSHRs merged back the cache-missed accesses, leading to the same number of accesses (Ⓑ) generated to the DRAM as that of in the baseline case. Consequently, it leads to the same correlation and information leakage as that of the baseline GPU.

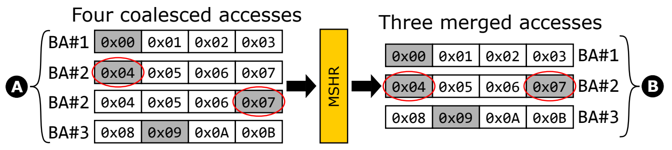


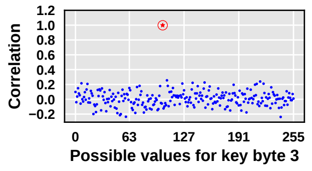
Fig. 7: Effect of MSHRs on the cache-missed coalesced accesses in RCoal scheme.

Effect of caches. The security of RCoal depends on the cache hit rates. For example, in the case of RCoal(32), if all accesses of a table lookup instruction are *always* cached, then all 32 accesses from the coalescing unit are served by the cache. Therefore, if the execution time remains constant due to the constant number of accesses to the cache, the attacker cannot establish the correlation between the number of baseline coalesced accesses and the execution time to reveal the private key. However, a perfect cache hit rate cannot be guaranteed for all the table lookup instructions across a large number of plaintext samples. Therefore, if the accesses of a table lookup instruction miss in the cache, the key byte can still be recovered with RCoal due to the access merging in MSHR as discussed earlier. To illustrate this point, Figure 8 shows the scatter plots for the first table lookup instruction of the last round with MSHRs and caches enabled. We note that the private key byte 3 corresponding to the first table lookup instruction can easily be recovered in both the RCoal scenarios.

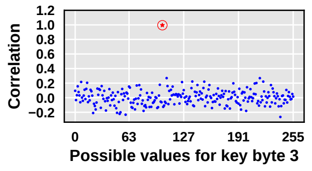
In summary, RCoal becomes vulnerable due to the access optimizations in MSHRs and caches.

C. Our Proposal and Goals

Our goal is to design a mechanism that reduces the performance overheads of RCoal while offering comparable security. To this end, we propose BCoal: a bucketing-based coalescing mechanism to address the primary performance-related shortcomings of RCoal discussed before. BCoal matches the number of coalesced accesses generated for a global memory load instruction per warp to one of the predetermined values (denoted as *buckets*). To match the number of accesses to one of the preset bucket sizes, we pad the real coalesced accesses from a warp with additional (padded) memory accesses. Since



(a) **RCoal(32) + MSHRs + Caches.**



(b) **RCoal(4) + MSHRs + Caches.**

Fig. 8: The presence of MSHRs and caches leads to successful recovery of one of the last round key bytes in RCoal(32) and RCoal(4). Plaintext has 32 lines.

the total numbers of accesses always match one of the bucket sizes, their overall variance decreases. Furthermore, as observed earlier for RCoal, MSHRs adversely affect the security by merging the redundant accesses after randomized coalescing. Therefore, the padding mechanism in BCoal is devised such that MSHRs cannot merge the real and padded accesses, thereby maintaining a very low variance in the resulting number of accesses. Additionally, the padding mechanism ensures that the real and padded accesses follow similar access merging and caching pattern, such that they affect the execution time at the same rate. Subsequently, the individual effects of real and padded accesses on the execution time are indistinguishable. Therefore, in BCoal-enabled GPU, the attacker will not be able to correlate the number of real coalesced accesses with the observed execution time. Consequently, the security offered by BCoal scheme against the correlation timing attacks remains intact even in the presence of MSHRs and caches. In summary, BCoal scheme presented in this work not only offers improved security but also incurs minimal performance degradation as compared to RCoal.

IV. Anatomy of Bucketing in GPUs

In this section, we first explain our general approach towards realizing a bucketing scheme and then explore the design challenges in meeting the bucketing requirements in the presence of MSHRs and caches. Finally, based on our analysis, we present our secure bucketing scheme – BCoal.

A. *Bucket Features*

Let us assume a system with n buckets and sizes of buckets to be: b_1,..,b_i, b_{i+1},...,b_n where $\forall i : b_i < b_{i+1}$. A predetermined number of coalesced accesses are generated per table lookup (load) instruction as per the bucket size. If a load instruction generates n number of coalesced accesses, where $b_i < n \leq b_{i+1}$, then additional accesses are padded such that the total number of coalesced accesses is equal to b_{i+1}. The number of buckets is selected to achieve the desired reduction in the variance of the number of coalesced accesses. For example, with only one bucket, the number of accesses generated is always equal to the size of that bucket, thus, reducing the variance to zero. As the number of buckets increases, the variance in the number of coalesced accesses increases due to the increased number of distinct possible values for the coalesced accesses. This leads to higher information leakage, however, also reduces the total number of additional padded accesses.

We revisit Figure 6(a) to select the bucket features for AES. We observe that the number of coalesced accesses during the AES encryption on GPU never exceeds 16. Therefore, we select the size of the bucket to be 16 as one of the options and denote the scheme as BCoal(16). With only one bucket of size 16 in the coalescing unit, the AES encryption will always generate 16 number of coalesced accesses to reduce their variance to 0. Consequently, the correlation between the number of real coalesced accesses and the execution time drops as well. However, with only one bucket, each (security-sensitive and security-insensitive) load instruction sends 16 accesses, leading to performance degradation (Section VI).

The performance of BCoal scheme can be further improved by adding multiple buckets of intermediate sizes. We propose to add one more bucket with size 1 because of the bi-modal distribution observed in Figure 6(a) and call this scheme BCoal(1, 16). The performance degradation in BCoal(1, 16) will be lower than in BCoal(16) because the coalesced accesses generated by instructions other than the table lookups (the first peak in Figure 6(a)) now fit into the added bucket. Furthermore, in BCoal(1, 16), as the bucket with size 1 does not affect the table lookup instructions, its effect on the security is minimum. We quantify all performance and security results in Section VI.

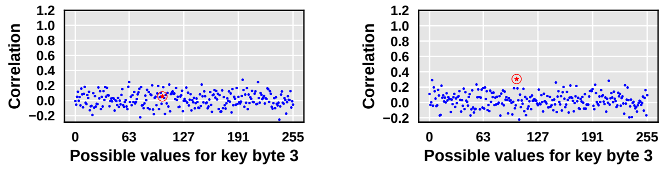
B. *Estimation of Number of Padded Accesses*

To generate an optimal number of padded memory accesses for bucketing, we first need to determine the number of real memory accesses generated for a load instruction of a warp. The number of real coalesced accesses generated by the load instruction is stored as the pending request count (PRC) in the coalescing unit [30]. By reading PRC, we determine the number of real memory accesses. Next, we compare the number of real memory accesses generated with the preset bucket values. If the number of real memory accesses does not match, then we generate a number of padded memory accesses equal to the difference between the next larger bucket value and the number of real memory accesses. For example, in BCoal(1,16), if the number of original memory accesses is 12, then we need to generate 4 extra memory accesses.

C. *Design Challenges in Generating Padded Accesses*

We consider the effect of MSHRs and caches on RCoal scheme while designing the padding mechanism for BCoal. In RCoal, the redundant accesses to the same block addresses were merged in MSHRs eliminating the security offered by randomized coalescing of accesses. Therefore, to meet the bucketing requirement, we must generate padded accesses to the *unique* block addresses. Consequently, all memory accesses originating from a warp, real and padded, have *unique* block addresses. The unique accesses for padding are generated *randomly* from an address range that is accessible to the AES CUDA application. In our case, the block address range spans over the five tables used for table lookups and the round keys used for each round, all saved in the DRAM.

To evaluate the resulting bucketing scheme, we first determine the possibility of key byte recovery in the absence of



(a) **Without MSHRs and caches.** (b) **With MSHRs and caches.**

Fig. 9: Evaluation of security offered by the bucketing scheme employing unique access padding mechanisms with one bucket of size 16. Plaintext has 32 lines.

MSHRs and caches. Figure 9a shows the scatter plots for the bucketing scheme employing padding via unique accesses in the absence of MSHRs and caches. We note that the correct value of the key byte cannot be recovered as the attacker fails to establish a correlation between the real number of accesses and the execution time. The low correlation is attributed to the constant number of accesses generated per table lookup instruction across the plaintext samples leading to the low variance in them.

From the above padding mechanisms employed in the bucketing scheme, we make the following observation:

Observation I: For the secure bucketing scheme, the block addresses of the padded accesses should be random and unique (that is, exclusive of the block addresses of the real and other padded accesses of the corresponding table lookup instruction).

Effect of MSHRs and caches. We evaluate the effect of MSHRs and caches on the security of above bucketing mechanism using the scatter plot in Figure 9b. We note that while the correlation and related key byte value leakage is low, the correct value of the key byte can still be recovered. The key byte value leakage is possible because the real and padded accesses affect the execution time at different rates due to their distinct merging and caching patterns.

The distinct access merging and caching patterns are caused because of the different block address ranges accessed by the real and padded accesses. In the above bucketing scheme, the padded accesses generated in each round access the same range of block addresses spread across the entire memory space of the AES CUDA application. In contrast to the padded accesses, in the last round, the real accesses target only the T_4 table elements. As the real accesses are confined to a narrower address space (only T_4 table elements) as compared to the padded accesses (entire application memory space), their respective merging and caching patterns are different. Therefore, the padded and real accesses affect the execution time at different rates. An attacker can then treat the effect of padded accesses on the execution time as noise and filter it out over a large number of plaintext samples to correlate the real accesses and the execution time to recover the private key. The effect of MSHRs and caches on the real and padded accesses leads to the following observation:

Observation II: The padded and real accesses should be homogeneous in terms of their respective probabilities of merging in MSHRs and caching.

D. BCoal: A Secure Bucketing Scheme

From the observations I and II recorded previously, we note that for a secure bucketing scheme to operate in the presence of MSHRs and caches, the padded accesses should have the following two characteristics: i) the block addresses of the padded accesses should be random and exclusive (unique) of the block addresses of the other accesses and ii) the padded accesses should follow the same merging and caching pattern as that of the real accesses.

Padding via Homogeneous Unique Accesses. The first property of the desired padding mechanism is met by ensuring that the block addresses of the padded accesses are random and unique across each security-sensitive load instruction. To enforce the second property, we recall the merging mechanism in MSHRs, where the accesses going to the same block addresses are merged together. Furthermore, the caching also works at the block address granularity. Therefore, to obtain similar merging and caching probabilities across all accesses, we restrict the block addresses of the padded accesses to the range of possible block addresses of the real accesses, thereby generating *homogeneous* unique accesses.[2]

During the AES execution, the table lookup instructions of the first nine rounds access first four tables, while for the last round only T_4 table is accessed. Therefore, to meet the bucketing requirements, the padding mechanism should restrict the block address range of the padded accesses to the block address range of the first four tables in DRAM during the first nine rounds, while to the block address range of T_4 table in DRAM during the last round. As the padding mechanism maintains similar merging and caching properties for the real and the padded accesses, the attacker cannot segregate their effects on the total execution time. Therefore, the attacker will fail to establish the correlation between the real number of coalesced accesses and the execution time, thereby failing to recover the key byte value. Furthermore, as all rounds of AES encryption are potentially vulnerable to timing attacks [32], BCoal is enabled for all ten rounds of AES.

In summary, we select the padding via homogeneous unique accesses for the BCoal bucketing scheme. We present the security and performance evaluation of the proposed BCoal scheme with MSHRs and caches enabled in Section VI.

V. Hardware/Software Overhead

In this section, we describe the implementation overhead of BCoal. We consider a generalized BCoal scheme, which targets a security-sensitive application with an arbitrary number of program sections. For example, the two program sections in AES are the first 9 rounds and the last round. The generated padded accesses have memory addresses that target respective program sections.

Storage overhead. The storage requirement is for keeping track of a) bucket sizes and b) the start/end addresses of the program sections. To store the buckets sizes, BCoal uses a

[2]This heuristic may have to be tuned for different applications based on their memory access pattern.

32-bit mask that covers all 32 possible number of coalesced accesses across a warp. The indices of the mask are set as per the BCoal configuration. For example, for BCoal(1, 16, 32), only 1st, 16th and 32nd bits are set. Next, BCoal maintains an address table – accessible by all SMs executing the security-sensitive application – to save the start and end 32-bit addresses of each program section. For an application with N program sections, the size of the table will be $(2N \times 32)$ bits. For AES with 2 program sections, the size of the table will be 128 bits and the total storage overhead is $128 + 32 = 160$ bits.

Address Generation. The generation of unique homogeneous accesses for padding follows three steps: a) determine the number of padded accesses needed, b) determine the unique homogeneous block addresses for the accesses, and c) generate the accesses. As noted in Section IV-B, the pending request count (PRC) in the memory coalescing unit (MCU) records the number of real accesses across a warp. Therefore, the number of padded accesses needed can be identified by comparing the size of a bucket with PRC. Since the maximum value of PRC (limited by the maximum possible number of coalesced accesses) and the maximum size of a bucket is 32, BCoal needs a 5-bit comparator.

The address range for each program section is known from the memory allocation and data copy operations executed at the start of a GPGPU application. This information can also be embedded in the load instructions. To generate padded accesses in the range of the program section under execution, BCoal uses a 32-bit random address generator.

VI. Analysis of Security & Performance

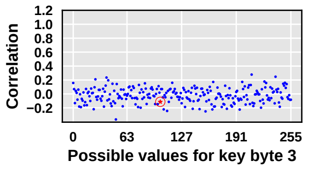
In this section, we first analyze the security of our proposed bucketing-based coalescing mechanism, BCoal, via experimental and theoretical analysis. Subsequently, we discuss the effects of the proposed mechanism on performance and data movement. We also compare BCoal with RCoal in terms of security, performance and data movement. Finally, we generalize our mechanism across a wide range of GPGPU applications.

All the results are collected on a cycle-level GPU simulator – GPGPU-Sim [22]. We assume the same number of samples as that of in the attack scenario [5] for plaintext with 32 lines. For plaintext with 64 lines, we use 1000 samples, the same number as needed for the successful attack, to evaluate the defense mechanism for a fair comparison.

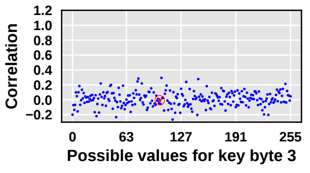
A. Experimental Analysis of Security

For the security evaluation of BCoal scheme in the presence of MSHRs and caches, we consider two configurations: i) default with one bucket of size 16 denoted as BCoal(16) and ii) performance efficient with two buckets of sizes 1 and 16 denoted as BCoal(1, 16). For each BCoal configuration, we plot a scatter plot as explained in Section II-D.

Plaintext with 32 lines. Figure 10 shows the scatter plots for BCoal scheme using plaintext with 32 lines with MSHRs and caches enabled. We note that the key byte recovery is not possible because of the low correlation between the number of accesses and the execution time. The low correlation can be explained as follows. With BCoal scheme, three scenarios can occur for a table lookup instruction. First, all accesses – real and padded – for the instruction are cached. In this case, the instruction always generates 16 accesses to the cache. Second, no accesses are cached, therefore, generating 16 DRAM accesses to the unique block addresses which MSHRs cannot optimize. In both scenarios, the number of accesses to the cache or DRAM remains constant leading to reduced correlation with the execution time. In the third case, a partial set of accesses of the instruction are either cached or merged in MSHR. Here, since the real and the padded accesses target the same block address range, their merging and caching probabilities are similar. Subsequently, the attacker cannot distinguish between the effects of the padded and real accesses on the execution time and fails to correlate the number of real coalesced accesses and the execution time. In conclusion, the attacker fails to recover the key byte in a BCoal-enabled GPU.

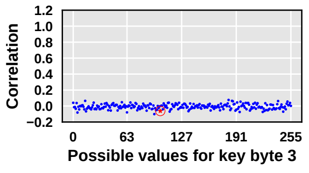


(a) BCoal(16)+MSHRs+Caches.

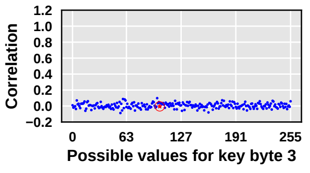


(b) BCoal(1, 16)+MSHRs+Caches.

Fig. 10: BCoal defense scheme against correlation attack for plaintext with 32 lines.



(a) BCoal(16)+MSHRs+Caches.



(b) BCoal(1, 16)+MSHRs+Caches.

Fig. 11: BCoal defense scheme against correlation attack for plaintext with 64 lines.

Plaintext with 64 lines. Figure 11 shows the scatter plots for BCoal scheme using plaintext with 64 lines with MSHRs and caches enabled. We note that the key byte recovery is not possible because of the low correlation between the number of accesses and the execution time. To understand the low correlation, we refer to the correlation timing attack described by Jiang et al. in [6] for the multi-warp case, where the attacker treats each warp individually executing a plaintext with 32 lines and chooses the warp with the highest number of coalesced accesses to recover the key. Therefore, the observations made for a single warp case hold true for the multi-warp case as well. Particularly, the attacker cannot correlate the number of real coalesced accesses and the execution time due to the low variance in the number of accesses, and the homogeneity between the real and padded accesses. Therefore, the attacker fails to recover the key byte value in the multi-warp scenario.

The experimental analysis concludes that BCoal-enabled GPU successfully mitigates the correlation timing attacks in single-warp and multi-warp scenarios.

B. Theoretical Analysis of Security

We present an analytical framework to analyze the security of AES. Before a formal analysis, we consider one instruction in the last round that accesses 12 unique memory block addresses before padding. When only one bucket is used (at 16), the 4 padded memory accesses are drawn from the same memory space as the 12 real requests. Hence, there is no information leakage. In general, we will shortly prove that when BCoal uses one bucket at 16, there is no information leakage.

When multiple buckets are used, say at 12 and 16, the attacker can infer if the number of real block addresses being accessed are up to 12 or between 12 and 16, which leaks some information. However, as we show next, the leakage in general is minimal, due to the randomized mapping from plaintext lines to warps. The randomized mapping obfuscates which plaintext lines share the same warp.

To quantify the leakage of BCoal, we note that threads across different warps are not synchronized and the longest warp execution time dominates the time measurement [6]. Hence, one of the warps, the dominant warp, will have true timing. Known attacks on multiple warps [6] analyze each warp and use the longest running (dominant) warp for correlation analysis to recover the AES private keys. So it is safe to focus on an *arbitrary* warp in the rest of the analysis. Moreover, we assume the padded and real accesses are homogeneous (as described in Section IV-B). Hence, their probabilities of merging in MSHRs and caching are identical.

To make a fair comparison with RCoal, we follow the analytical model and assumptions of RCoal [5]. Futher, we target an arbitrary last-round key byte k and assume that U is the number of real accesses for the lookup of last round table, T_4, with respect to the key byte k, from the dominant warp. Following RCoal [5], we estimate the number of plaintext samples required to successfully recover an AES key byte, S, as

$$S \propto \Big(\frac{\mu(U\times\widehat{U})-\mu(U)\mu(\widehat{U})}{\sigma(U)\sigma(\widehat{U})}\Big)^{-2} \qquad (1)$$

where $\widehat{U}$ is the number of coalesced accesses when the guessed key byte is identical to k, μ and σ are the mean and standard deviation of a random variable respectively.

We first prove BCoal leaks no information with one bucket.

Lemma 1. *When BCoal only uses one bucket at 16, the needed samples to break AES is infinite.*

Proof. With only one bucket, $P(\widehat{U}=16|U=u)=1$ for any u. Hence, $\mu(\widehat{U})=16$ and $\mu(U\times\widehat{U})=\sum_u P(u)\mu(U\times\widehat{U}|U=u)=16\sum_u u\times P(u)=16\mu(U)$. Hence $S=(0)^{-2}=\infty$. $\square$

When the number of buckets is more than one, the computation is more involved. To simplify the analysis, we further make a conservative assumption that an attacker may directly observe the unpadded memory blocks in the following analysis. Therefore, $\mu(\widehat{U})=\mu(U), \sigma(\widehat{U})=\sigma(U)$.

In AES, the lookup table relevant to key byte k has 16 unique memory block addresses. With sufficiently random plaintexts and a warp with 32 threads, each thread accesses one of 16 memory block addresses in a uniform way. Hence, the number of unique block addresses U, obeys the following distribution: $P(U=i)=\frac{1}{16^{32}}\frac{16!}{(16-i)!}\left\{{32 \atop i}\right\}$, where $\left\{{32 \atop i}\right\}$ denotes the Stirling number of the second kind. Here, $\left\{{32 \atop i}\right\}$ represents the ways of partitioning 32 threads into i non-empty subsets; $\frac{16!}{(16-i)!}$, i-permutations of 16, represents the ways of forming i non-empty subsets from 16 memory block addresses. From this distribution, we can compute both $\mu(U)$ and $\sigma(U)$ by their definitions.

To compute $\mu(U\times\widehat{U})$, we note that due to the random mapping from plaintext lines to warps, U and $\widehat{U}$ only depend on the frequency of accessing the 16 memory block addresses among the 64 lines of plaintext, which is defined as follows.

Definition 1. *For 16 memory blocks and 64 plaintext lines, the* frequency set *of all possible accesses to the block addresses are*

$$\mathscr{F}=\{(f_1,\dots,f_{16}) \mid f_1+\dots+f_{16}=64\}$$

where $f_i\in\mathscr{F}$ represents the frequency of accessing the i-th memory block address among the 64 plaintext lines.

Given $F\in\mathscr{F}$, $\mu(U|F)=\sum_{f_i\in F}\mu(\mathbf{1}_{\text{block i is accessed}}|f_i)$, where $\mathbf{1}_{\text{block i is accessed}}$ is an indicator random variable that has value 1 if block address i is being accessed in the dominating warp. Given f_i accesses to block address i, the probability that it is accessed in the dominating warp is $(1-C^{64-32}_{f_i}/C^{64}_{f_i})$, where C^m_n denotes the binomial coefficient. Hence,

$$\mu(U|F)=\sum_{f_i\in F}1-C^{32}_{f_i}/C^{64}_{f_i}$$

Given $F\in\mathscr{F}$, U and $\widehat{U}$ are independently and identically distributed. Hence,

$$\mu(U\times\widehat{U})=\sum_{F\in\mathscr{F}}P(F)\mu(U|F)^2=\sum_{F\in\mathscr{F}}P(F)\Big(\sum_{f_i\in F}1-\frac{C^{32}_{f_i}}{C^{64}_{f_i}}\Big)^2$$

Here, $P(F)$ is the probability of seeing the frequency vector F. Among all 16^{64} combinations of memory accesses from 64 threads, $C^{64}_{f_1}C^{64-f_1}_{f_2}\cdots C^{64-\sum_{1\le j\le 15}f_j}_{f_{16}}=\frac{(64)!}{\Pi_{f_i\in\mathscr{F}}f_i!}$ match F. Hence, we have $P(F)=\frac{(64)!}{\Pi_{f_i\in\mathscr{F}}f_i!}\times\frac{1}{16^{64}}$.

Putting all pieces together, we use a Python script to compute the correlation and normalized the sample size needed for a successful attack, similar to the RCoal analysis [5]. The results are summarized in Table II. We note that with 1 bucket, BCoal rules out leakage entirely. With multiple warps, its security is comparable with RCoal(4), the best of the RCoal schemes. Note that the results of RCoal in Table II only applies when MSHR and caches are *disabled*. But with homogeneous padded

TABLE II: Security Analysis. S denotes the normalized number of samples required to successfully recover an AES key byte [5].

Schemes	Correlation ρ	(normalized) S
RCoal(4)	0.15	42×
RCoal(32)	0.00	∞
BCoal(16)	0.00	∞
BCoal(1,16)	0.16	37×

and real accesses, the results of BCoal also applies even if MSHR and caches are *enabled*.

In summary, this theoretical security analysis demonstrates that when MSHR and caches are disabled, both RCoal and BCoal schemes provide significant security against the correlation timing attack. However, if the MSHRs and caches are enabled, RCoal becomes vulnerable due to the access merging and caching as illustrated in Figure 8. In contrast to RCoal, BCoal has high security even in the presence of MSHRs and caches as shown both in Table II and in Section VI-A.

C. Experimental Analysis of Performance

To evaluate the performance and scalability of BCoal scheme against RCoal, we plot the execution time and number of DRAM accesses in Figure 12 for plaintext with 32, 64 and 1024 lines. We first demonstrate the effect of different coalescing strategies in BCoal and RCoal by comparing them in the absence of MSHRs and caches in Figure 12a. We note that the number of DRAM accesses increases sharply with the plaintext size in RCoal as compared to BCoal due to the inefficient access coalescing in RCoal. Consequently, RCoal suffers severe performance degradation as compared to BCoal as the plaintext size increases.

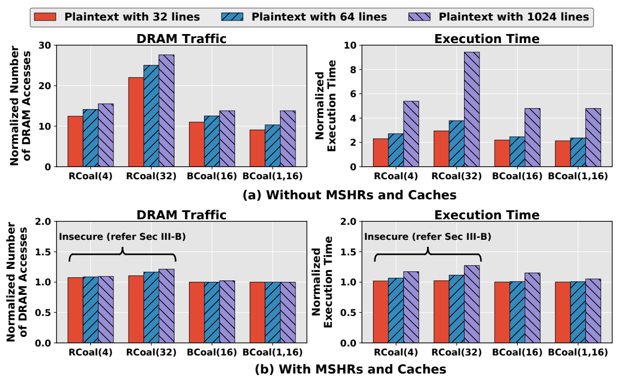


Fig. 12: Performance of BCoal for different plaintext sizes. All results are normalized to the baseline GPU.

Figure 12b demonstrates the effect of MSHRs and caches on the performance of BCoal and RCoal. Both schemes show a significant reduction in the DRAM traffic leading to reduced performance degradation. However, in the presence of MSHRs and caches, RCoal is insecure (Section III-B) and BCoal is secure (Section VI-A and VI-B). For BCoal, the performance degradation is limited to 5% and 15% for BCoal(1, 16) and BCoal(16), respectively. In summary, the performance of BCoal (with MSHRs and caches) scales well with the plaintext size as opposed to secure RCoal (without MSHRs and caches).

D. Evaluating BCoal on Other Applications

We evaluate BCoal on a wide range of applications from various suites such as CUDA-SDK (C) [33], Rodinia (R) [34], Lonestar (L) [35], Mars (M) [36], Shoc (S) [37] and Polybench (P) [38]. For these applications, we evaluate only the performance of BCoal, as the bucketing driven reduced variation in the number of coalesced accesses ensures improved security. The address range of the padded accesses is spread over the entire memory space of the respective application. We examine the effects of the number and sizes of buckets on the application performance using Figure 13. The MSHRs and caches are enabled for the evaluation.

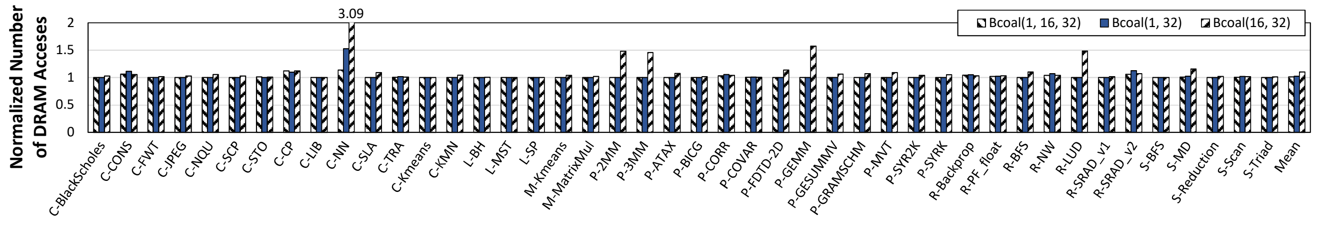
Number of buckets. In Figure 13, the first two configurations of BCoal, BCoal(1, 16, 32) and BCoal(1, 32), demonstrate the effect of the number of buckets on various applications. Both configurations have a bucket of size 1 to reduce the DRAM traffic in applications that exhibit perfect coalescing (i.e, all threads in a warp are served by a single cache block at a given time). We notice that most applications are unaffected by the number of buckets, as they can leverage the bucket of size 1 through good coalescing profiles.

In `C-CONS` and `C-NN`, the number of DRAM accesses increase in BCoal(1, 32) as the number of coalesced accesses between 2 to 31 are padded to meet the bucket 32. The increased number of accesses in combination with high cache-misses results in increased DRAM traffic leading to increased performance degradation. In `C-TRA`, `P-CORR` and `P-COVAR`, although the number of DRAM accesses does not change drastically, the execution time increased in BCoal(1, 32) over BCoal(1, 16, 32). The increase in execution time is attributed to the increase in the number of L1 cache accesses in BCoal(1, 32) as it lacks the bucket of size 16. The increased L1 accesses, even if cached (thus leading to fewer DRAM accesses), are satisfied serially thereby increasing the execution time.

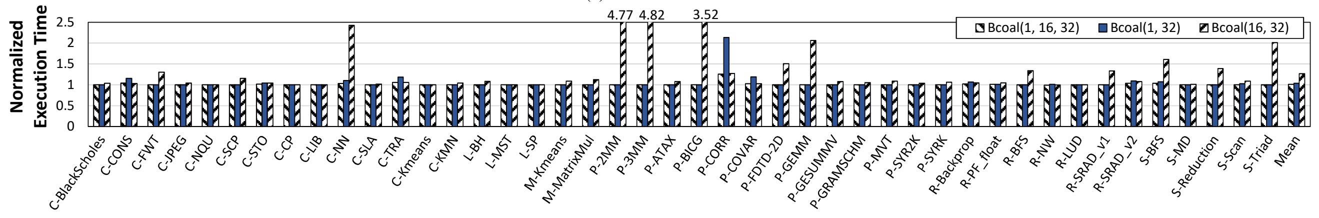
Sizes of buckets. BCoal(1, 32) and BCoal(16, 32) demonstrate the effect of bucket sizes on various applications. We noticed that the performance degradation is severe for BCoal(16, 32) compared to BCoal(1, 32) due to the increased number of DRAM accesses in BCoal(16, 32). In BCoal(16, 32), the smallest bucket size is 16, therefore all applications, even the ones with good coalescing profiles, generate at least 16 DRAM accesses for each memory access instruction. Subsequently, the number of DRAM accesses increase resulting in increased performance degradation.

In summary, we observe that the application performance is more affected by the sizes of buckets than the number of buckets. A careful bucket size selection can reduce the number of padded requests thereby reducing the overall data movement.

A Generic BCoal configuration. From Figure 13, we note that BCoal(1, 16, 32) configuration results in only 1.15% average performance loss. The security and performance of AES with BCoal(1, 16, 32) is identical to BCoal(1, 16) because the bucket of size 32 in BCoal(1, 16, 32) is never used as the baseline number of coalesced accesses never exceed 16 as shown in Figure 6a. Therefore, BCoal(1, 16, 32) can be widely adopted as it offers good security at a minimal performance loss. However, for optimal security and performance tradeoff, a user can perform application-specific offline profiling of coalesced accesses (discussed in Section III) to determine appropriate bucket features.



(a) **DRAM Traffic.**



(b) **Execution Time.**

Fig. 13: Performance Evaluation of BCoal on GPGPU applications with MSHRs/caches enabled. Results are normalized to the baseline GPU.

VII. Related Work

In this section, we highlight the prior works that are the most relevant to this paper.

Attacks. Implementations of cryptographic systems on CPUs are vulnerable to timing attacks. Several AES implementations contain key-dependent memory accesses, which eventually affect the status of the data cache. Via cache-probing technique, an attacker can quickly recover the entire private key of AES and RSA by measuring the execution time of either a cryptographic algorithm (e.g., [32], [39]–[42]) or his/her own application if the data or instruction cache is shared (e.g., [42]–[45]). On GPUs, Jiang et al. [6] demonstrated a novel complete AES key recovery timing attack that exploits the coalescing features on a commercial GPU architecture (discussed in Section II-D). They also developed a new fine-grained timing channel caused by shared memory bank conflicts in GPUs [21]. Wang et al. [46] developed partial attacks against RCoal [5] focusing on the configurations with high variance in the number of coalesced accesses. Our BCoal mechanism further reduces the variance making it a much stronger defense.

Defense mechanisms. Several hardware-based defense mechanisms have been proposed in the context of CPUs [13]–[15], [47]–[50]. However, those mechanisms have been shown to work only for cache-based timing attacks and not for GPU coalescing-related vulnerabilities. The memory traffic shaping schemes to mitigate the timing attacks in CPUs have been extensively explored [51]–[53]. With the help of fake/dummy access generation mechanism, these schemes enforce the memory traffic to follow either a constant rate or a predetermined distribution over a time epoch. These schemes differ from BCoal in two ways. First, BCoal works at a finer instruction-level granularity to shape the memory traffic. The single-instruction multiple-thread (SIMT) execution model of GPUs allows parallel thread memory access generation across a warp, which is leveraged by BCoal to estimate and generate padded accesses for each sensitive instruction. Second, BCoal ensures that the real and padded accesses are to the same memory space, which helps in making their individual effects on execution time similar. This makes it harder for the attacker to distinguish padded accesses from the real accesses.

Lin et al. [54] proposed new software-based mechanisms specific to AES for reducing the information leakage due to coalescing units. On the other hand, BCoal is a generic hardware-based coalescing mechanism applicable to all security-sensitive GPGPU applications that are vulnerable to coalescing-based correlation timing attacks. This also makes BCoal complementary to other software-based implementations of cryptographic workloads. Köpf et al. [55] ensures that the execution time matches one of the discrete bucket values, while BCoal ensures the number of memory accesses generated per load instruction conform to a predefined set of values, that is buckets. Further, buckets in the prior work [55] assumes input blinding for a tight leakage bound. In BCoal, we utilize the inherent parallelism in GPUs to randomize the mapping from inputs to threads, achieving a similar blinding effect for arbitrary applications.

VIII. Conclusions

We propose a bucketing-based coalescing scheme (BCoal) to thwart the coalescing-based correlation timing attack without incurring high performance overhead. The key insight is to redesign GPU memory coalescing such that it always issues a pre-determined number of memory accesses (called buckets). Our modified coalescing unit generates additional memory accesses (if necessary) along with the real accesses to match the bucket requirements. These additional padded accesses reduce the variance in the total number of coalesced accesses to significantly enhance the security. BCoal carefully generates padded accesses such that they have similar caching/merging probability as that of the real accesses. Such a mechanism significantly helps in retaining the security even in the presence of the MSHRs and caches. In conclusion, we believe that BCoal addresses the memory coalescing related vulnerability in GPUs while incurring low performance overhead.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and the members of Insight Computer Architecture Lab at William & Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) grant #1717532 and a summer research grant from William & Mary. This work was performed in part using computing facilities at William & Mary.

REFERENCES

[1] NVIDIA, "Parabricks." [Online]. Available: https://blogs.nvidia.com/blog/2018/09/05/parabricks-genomic-analysis/
[2] NVIDIA, "Computational finance." [Online]. Available: https://www.nvidia.com/en-us/gtc/topics/finance/
[3] W. Hua, Z. Zhang, and G. E. Suh, "Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks," in *DAC*, 2018.
[4] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *CCS*, 2018.
[5] G. Kadam, D. Zhang, and A. Jog, "RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques," in *HPCA*, 2018.
[6] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *HPCA*, 2016.
[7] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted Execution Environments on GPUs," in *OSDI*, 2018.
[8] G. Liu, H. An, W. Han, G. Xu, P. Yao, M. Xu, X. Hao, and Y. Wang, "A program behavior study of block cryptography algorithms on GPGPU," in *FCST*, 2009.
[9] T. Cheneau, A. Boudguiga, and M. Laurent, "Significantly improved performances of the cryptographically generated addresses thanks to ECC and GPGPU," *computers & security*, vol. 29, 2010.
[10] S. Neves and F. Araujo, "On the performance of GPU public-key cryptography," in *ASAP*, 2011.
[11] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL Acceleration with Commodity Processors," in *NSDI*, 2011.
[12] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *ACSAC*, 2006.
[13] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.
[14] Z. Wang and R. B. Lee, "A Novel Cache Architecture with Enhanced Performance and Security," in *MICRO*, 2008.
[15] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *MICRO*, 2014.
[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.
[17] M. K. Qureshi, "New Attacks and Defense for Encrypted-address Cache," in *ISCA*, 2019.
[18] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are Coherence Protocol States Vulnerable to Information Leakage?" in *HPCA*, 2018.
[19] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on GPGPUs," in *MICRO*, 2017.
[20] Q. Xu, H. Naghibijouybari, S. Wang, N. B. Abu-Ghazaleh, and M. Annavaram, "GPUGuard: mitigating contention based side and covert channel attacks on GPUs," in *ICS*, 2019.
[21] Z. H. Jiang, Y. Fei, and D. Kaeli, "A Novel Side-Channel Timing Attack on GPUs," in *VLSI*, 2017.
[22] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[23] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.
[24] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.
[25] F. P. Miller, A. F. Vandome, and J. McBrewster, *Advanced Encryption Standard*. Alpha Press, 2009.
[26] O. Harrison and J. Waldron, "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units," in *CHES*, 2007.
[27] K. Iwai, T. Kurokawa, and N. Nisikawa, "AES Encryption Implementation on CUDA GPU and Its Analysis," in *ICNC*, 2010.
[28] N. Nishikawa, K. Iwai, and T. Kurokawa, "High-Performance Symmetric Block Ciphers on CUDA," in *ICNC*, 2011.
[29] Q. Li, C. Zhong, K. Zhao, X. Mei, and X. Chu, "Implementation and analysis of AES encryption on GPU," in *HPCC-ICESS*, 2012.
[30] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.
[31] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up GPU warps by reducing memory pitstops," in *HPCA*, 2015.
[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," *CT-RSA*, 2006.
[33] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples
[34] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.
[35] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IISWC*, 2012.
[36] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.
[37] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," in *GPGPU*, 2010.
[38] L.-N. Pouchet, "Polybench: the polyhedral benchmark suite," 2012. [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/polybench/
[39] D. J. Bernstein, "Cache-timing attacks on AES," cr.yp.to/papers.html#cachetiming, 2005.
[40] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *CHES*, 2006.
[41] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, "Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs," in *CT-RSA*, 2010.
[42] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games—Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.
[43] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *USENIX Security*, 2014.
[44] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," in *RAID*, 2014, pp. 299–319.
[45] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *CCS*, 2014.
[46] X. Wang and W. Zhang, "Cracking Randomized Coalescing Techniques with An Efficient Profiling-Based Side-Channel Attack to GPU," in *HASP*, 2019.
[47] D. Page, "Partitioned cache architecture as a side-channel defense mechanism," in *Cryptology ePrint Archive, Report 2005/280*, 2005. [Online]. Available: http://eprint.iacr.org/2005/280.pdf
[48] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *ASPLOS*, 2014.
[49] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *ASPLOS*, 2015.
[50] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks," in *ISCA*, 2017.
[51] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, "Camouflage: Memory traffic shaping to mitigate timing attacks," in *HPCA*, 2017.
[52] C. W. Fletchery, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs," in *HPCA*, 2014.
[53] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory Defenses for Memory Bus Side Channel," in *ISCA*, 2017.
[54] Z. Lin, U. Mathur, and H. Zhou, "Scatter-and-Gather Revisited: High-Performance Side-Channel-Resistant AES on GPUs," in *GPGPU*, 2019.
[55] B. Köpf and M. Dürmuth, "A Provably Secure And Efficient Countermeasure Against Timing Attacks," in *CSF*, 2009.