IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 25 September 2022, accepted 16 October 2022, date of publication 21 October 2022, date of current version 26 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3216325

RESEARCH ARTICLE

GhostLeg: Selective Memory Coalescing for Secure GPU Architecture

JONGMIN LEE[1], SEUNGHO JUNG[1], TAEWEON SUH[1], (Member, IEEE), YUNHO OH[2], (Member, IEEE), MYUNG KUK YOON[3], (Member, IEEE), AND GUNJAE KOO[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Korea University, Seoul 02841, South Korea
[2]School of Electrical Engineering, Korea University, Seoul 02841, South Korea
[3]Department of Computer Science and Engineering, Ewha Womans University, Seoul 03760, South Korea

Corresponding author: Gunjae Koo (gunjaekoo@korea.ac.kr)

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2019-0-00533, Research on CPU Vulnerability Detection and Validation/IITP-2022-2020-0-01819, ICT Creative Consilience Program/No. 2021-0-02068, Artificial Intelligence Innovation Hub), in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1C1C1012172), and in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2022R1C1C1011021).

ABSTRACT Architectural considerations for secure executions are getting more critical for GPU since popular security applications and libraries have been ported to a GPU domain to rely on GPU's massively parallel computations. Recent studies disclosed the security attack models that exploit GPU's architectural vulnerabilities to leak the secret keys of AES. The attack models exploit the high correlations between the execution time of a kernel and the number of memory requests generated from memory coalescing. Thus the prior architectural defenses provide secure executions by randomizing or restricting the memory coalescing from load warps. However, those defense approaches result in significant performance overhead since memory coalescing is an essential feature for improving the performance of GPU. In this paper, we propose GhostLeg, an efficient architectural defense approach against correlation-based GPU security attacks. GhostLeg selectively applies secure executions for load warps to minimize performance overhead induced by concealing memory coalescing behavior. Our analysis of AES reveals that only the load warps whose index addresses are influenced by secret keys are vulnerable to security attacks. In order to minimize the performance overhead by secure executions, GhostLeg pinpoints the load warps that require secure executions based on the class of a source register. The *secure* flag assigned to each register can be set by propagation from non-deterministic user data (GhostLeg-ND) or a specific directive marked by programmers (GhostLeg-Key). Our evaluation shows that GhostLeg guarantees secure executions against the correlation-based attacks and GhostLeg-ND exhibits 54.7% higher performance compared to the state-of-the-art GPU defense solution.

INDEX TERMS GPU, secure architecture, security attack, memory coalescing.

I. INTRODUCTION

Secure graphics processing unit (GPU) architecture is critical for building trusted computing systems that rely on GPU's high computing performance. GPUs exploit massive thread-level parallelism (TLP) to boost the performance of parallel applications. Hence, various kinds of applications in a CPU domain have been ported to a GPU domain to rely on GPU's massively parallel computation capability. Recently GPUs are deployed to run popular security libraries which demand heavy computations. For instance, GPU's software execution model is suitable to encrypt/decrypt a huge volume of plain text data in parallel, thus using GPUs we can dramatically reduce the execution time taken by

The associate editor coordinating the review of this manuscript and approving it for publication was Zheng Yan.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

encryption/decryption processes. As such, GPUs have become an essential part of implementing efficient trusted computing environments.

Architectural considerations for defending against software-based security attacks are crucial for secure processor design since lots of security attacks exploit architectural vulnerabilities and/or side-channels in processor hardware to leak secret data. Some researchers revealed that essential architectural features in modern CPUs can be exploited for leaking private or protected data. For instance, Spectre attacks can read out any secret data exploiting speculative executions and cache timing side-channels, which are essential architectural ideas in a CPU domain [1], [2], [3], [4], [5]. Security attacks often target popular encryption/decryption libraries (e.g. AES and RSA algorithms) which are designed for providing data confidentiality. It is because such algorithms use *secret* keys for encrypting/decrypting private data thus attackers can spy on private data if the secret keys are exposed. Previous studies demonstrated attackers can read out secret keys of AES and RSA algorithms by exploiting side-channels in CPUs [6], [7], [8], [9], [10], [11]. In a GPU domain, researchers revealed that the secret keys of the AES algorithm can be decoded by calculating correlations between the number of memory requests by the estimated key value and the measured execution times of the parallel AES algorithm [12]. Such correlation-based attack exploits GPU's unique architectural feature, called *memory coalescing*, in load/store units. Namely, the memory requests generated from the dozens of threads in the same thread execution group (called a warp or a wavefront) are merged into a single memory transaction if the addresses of the multiple memory requests can be included in the address range of a single memory access. GPU performance can be improved if multiple memory requests are merged into a small number of transactions by the memory coalescing. Therefore, the execution time (i.e. performance) of a certain GPU application is highly correlated with the number of generated memory transactions. Consequently, attackers can estimate secret keys by monitoring the execution times of an encryption application if the number of generated memory requests is determined by the secret keys.

A straightforward defense solution against the correlation-based attack is nullifying the outcomes from the memory coalescing in load/store units. However, such an approach results in significant performance drops since memory coalescing is an essential feature for improving the performance of GPU. Previous studies propose defense approaches that randomize the memory coalescing mechanism or restrict the possible number of generated memory transactions [13], [14]. However, those solutions lead to significant performance loss since such approaches invalidate the benefits of memory coalescing.

In order to tackle the performance issue of secure GPU architectures, we propose GhostLeg, an efficient defense approach against the correlation-based attacks. We first investigate the CUDA version of AES to reveal a specific class of load warps is vulnerable to the correlation-based security attacks. We call such type of load warps as *non-deterministic* loads since the number of memory requests from the load warp is computed from *secret* data. As the execution time of AES is proportional to the number of generated memory transactions, attackers can figure out the secret data by probing the execution latencies of AES for many plain text inputs. Our motivation experiments exhibit prior defense solutions result in significant performance drops for general-purpose applications since those solutions create too many padded memory requests from all load warps to conceal the behavior of memory coalescing. In order to minimize such unnecessary dummy memory requests, GhostLeg identifies the load warp that requires secure executions by checking the class of the source register which is used for computing the addresses of the load warp. In this paper, we propose two different methods that decide the classes of register data. Namely, when a register is updated by loaded data, GhostLeg can set the class of the target register automatically or by using a directive specified by programmers. Since GhostLeg can effectively choose the load warps that require secure executions, GhostLeg minimizes the performance overhead caused by secure executions for defending against the security attacks.

The remainder of this manuscript is organized as follows. We introduce GPU's memory hierarchy architecture and the correlation-based attack mechanism in Section II. We investigate the performance issues of architectural defense approaches in Section III. The main idea and the architecture of the proposed scheme are described in Section IV. Evaluation results are exhibited in Section V. We discuss the related work in section VI. We conclude in Section VII.

II. BACKGROUND

In this section, we describe the architecture of the GPU load/store units and the memory subsystem which are optimized for regular memory access patterns generated from multiple threads. Based on GPU's unique memory coalescing mechanism, the actual number of memory transactions is determined by data index arrays of multiple threads. Then we briefly describe how attackers can decode secret keys from GPU AES algorithms by exploiting correlation-based attacks.

A. GPU MEMORY SUBSYSTEM ARCHITECTURE

GPU architecture is designed for supporting massive thread-level parallelism. In order to run hundreds or thousands of concurrent threads, a GPU equips hundreds of *simple* pipelined cores that can execute basic integer or floating-point instructions. In a GPU, for efficient parallel executions, dozens of threads that share the same instructions are grouped into an execution group, called a warp (by NVIDIA) or a wavefront (by AMD). Namely, a GPU schedules the grouped threads (i.e. a warp) to dozens of simple cores in a streaming multiprocessor (SM). Please note that in this paper we use NVIDIA's terminology to describe GPU architecture. As AMD GPU hardware is organized similarly to NVIDIA's

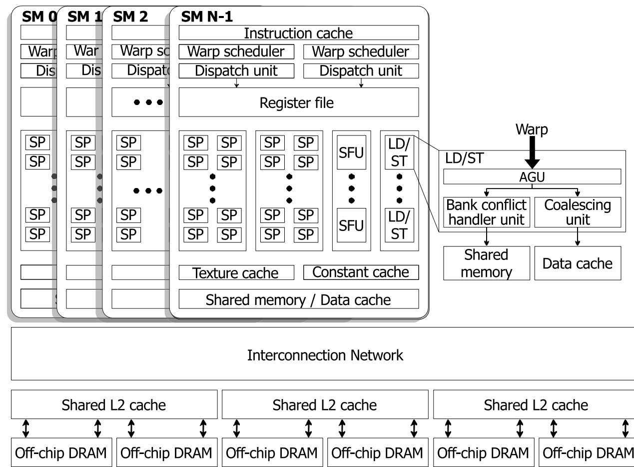

FIGURE 1. GPU architecture.

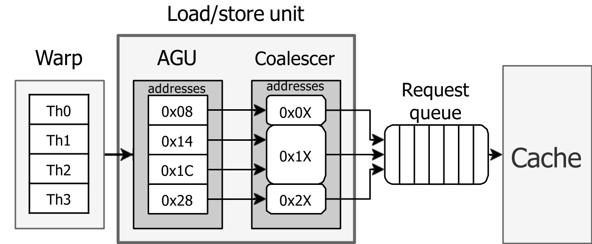

FIGURE 2. Coalescer and the number of memory requests.

generic GPU architecture, our methodology can be obviously applied to AMD GPUs also. As shown in Figure 1, a single SM includes dozens of processing cores (streaming processors, SPs), special function units (SFUs), and load/store units (LSUs). Each SP in an SM is coupled with the corresponding thread in a warp, thus threads in a warp can be scheduled every cycle if a warp executes basic integer/floating-point instructions and all required operands are ready.

Unlike the basic integer/floating-point instructions that rely on multiple processing units, GPU load/store instructions are serialized since memory has limited hardware resources that can handle read/write requests. Namely, the multiple load/store instructions from dozens of threads in a single warp cannot be handled concurrently in GPU memory subsystem. However, GPU's performance will extremely go down if all memory transactions from individual threads are handled one by one in the memory subsystem. In order to avoid performance drops by serialized memory requests, GPU load/store units support memory coalescing. The load/store unit merges multiple memory requests from threads in a warp to generate a single memory transaction if the request addresses can be included within the predefined address range (e.g. 64B or 128B size). GPUs can effectively reduce the number of memory transactions using memory coalescing if threads within a single warp exhibit regular memory access patterns. Note that a GPU groups threads which have similar thread indexes into the same warp and, in many cases, each thread accesses arrays using an index number computed from block/thread indexes [15].

Figure 2 depicts how memory coalescing works in GPUs using a simple example. In this example, we assume a single warp includes four threads and a single memory transaction is aligned to an address range of 16 bytes. When the warp executes a memory instruction, an address generation unit (AGU) in the load/store unit computes the request address of each thread. In the example, the memory requests from threads 1 and 2 can be merged into a single memory request of address `0x10` since the addresses from treads 1 and 2 are in the range from `0x10` to `0x1F`. Hence, the memory coalescing unit (i.e. coalescer in Figure 2) generates three memory transactions out of four memory requests from the threads in the warp. In this paper, we call an individual memory request from each thread a per-thread memory request (PTMR) and the memory requests merged by the memory coalescer of the load/store unit coalesced memory requests (CMRs) respectively. Obviously, the number of CMRs is influenced by the memory access patterns of PTMRs. The latency of load/store units can be increased as more memory transactions are generated after memory coalescing. Hence, the total execution time of GPU applications can be influenced by the degree of memory coalescing.

B. CORRELATION-BASED SECURITY ATTACKS ON GPU

Researchers disclosed secret keys of Advanced Encryption Standard (AES) on GPU can be extracted by exploiting correlation-based attacks [12]. Such security attacks exploit the observations that the execution time of an application is proportional to the generated CMRs when the application runs on GPUs. AES ported to a GPU domain implements an electronic codebook (ECB) mode since AES's encryption/decryption processes can be performed in parallel for separated plain text blocks [16]. Unlike other implementations such as cipher block chaining (CBC), AES which implements the ECB mode can exploit massive thread-level parallelism of GPU. Namely, each thread executes the AES kernel code for a single plain text block and a GPU runs hundreds of concurrent threads from the AES kernel. As explained in Section II-A, dozens of threads (32 threads for NVIDIA GPU) are grouped into a warp and all threads within a warp execute the same instructions.

The AES encryption kernel performs block cipher as follows. In the initial round, 16-byte plain text is added (XORed) with the first round encryption key (called an *AddRoundKey* step) to generate a 4×4 array. Then in every round the AES kernel performs *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* steps to generate a 4×4 array, which is the result of the corresponding round. In order to reduce computation burdens, AES utilizes the pre-calculated lookup tables (called T-tables) that combine *SubBytes*, *ShiftRows*, and *MixColumns* operations. Namely, in each round the AES kernel performs the table lookups using the 16-byte result from the preceding round, then the generated

4×4 array is added with the round key expanded from the initial encryption key to generate a 16-byte round result. In the last round AES does not perform *MixColumns*, thus the AES kernel employs a different lookup table only for the last round. Since AES utilizes the pre-calculated lookup tables for fast computations, the threads within the same warp share the same T-tables which are accessed by round result arrays combined with round keys. Note that memory transactions in a warp coalesce in GPU load/store units thus the number of CMRs varies by the AES keys. This is the key observation exploited by the correlation-based GPU security attacks.

Now we describe the attack process of the correlation-based security attacks. The last round of AES encryption is simply reversible since the last round does not include mixing operations (i.e. *MixColumns*) [17]. Thus attackers can easily compute the number of CMRs that access T-tables in the last round by inverting the last round operations using ciphertext blocks and estimated last-round keys. Note that T-tables are aligned in GPU memory space and the warps from the AES kernel access T-tables with a fixed size of coalescing segment. For each ciphertext block, attackers calculate the number of CMRs in the last round by applying an estimated last-round key. Note that there is a total of 256 possible cases for a 1-byte last-round key. Moreover, attackers measure the execution time for generating the ciphertext blocks using AES encryption on a GPU. It is revealed that the execution time of an AES encryption kernel on GPU is proportional to the number of CMRs generated in the last round of AES [12]. With lots of ciphertext samples, attackers compute correlations between the measured execution times and the number of CMRs by the estimated keys. We can get 256 correlation values for 256 possible cases for a 1-byte last-round key. Among the 256 possible keys, the key that exhibits the highest correlation is the last-round encryption key. Once the last-round key is leaked, attackers can reversely compute the original encryption key.

Since the correlation-based security attacks exploit the observation that the number of CMRs which access AES T-tables is influenced by encryption/decryption keys, the straightforward defense approach against the attacks is to disable memory coalescing in GPU load/store units. However, such an approach drops the performance of GPU significantly. GPU performance is significantly influenced by the degree of memory coalescing since the actual intensity of memory transactions in GPU is determined by the number of CMRs [18]. Several researchers proposed a defense approach that can randomize the number of CMRs from a warp by assigning coalescing thread groups randomly [13]. However, such an approach cannot guarantee secure AES executions since uncoalesced memory transactions can be merged in miss status handling registers (MSHRs) if the transactions are missed in the cache. Namely, the multiple memory requests generated from the separated coalescing groups can be merged as a single outstanding request to the lower-level cache if these multiple requests are assigned in a single cache line range. In this case, attackers can formulate correlation-based attacks using the intensity of outstanding requests biased by encryption keys. In order to guarantee secure computing environments for AES on GPU, the CMRs generated from the randomized coalescing groups need to bypass cache hierarchy, however, it will degrade the performance of general-purpose GPU applications significantly. Another defense scheme is to restrict the possible coalescing levels from the memory coalescer [14]. Instead of disabling the memory coalescing entirely, the GPU that employs this solution, called BCoal, generates CMRs as many as the upper bound of a range of CMR counts [14]. For instance, BCoal produces 4 more memory transactions to generate a total of 16 requests if the original count of CMRs is 12 and the upper bound of the coalescing range is set as 16. This solution can effectively defend against the correlation-based attacks, however, we cannot avoid significant performance drops resulting from increased memory transactions. We will explore this aspect further in the next section.

III. PERFORMANCE ISSUES OF SECURE GPU

Even though the existing architectural defense schemes can provide secure execution environments for AES on GPU, those solutions drop the performance of GPU significantly. Note that such architectural approaches also downgrade the performance of general-purpose GPU applications which do not require secure executions. In this section, we analyze the critical performance issues of the state-of-the-art defense solution against the correlation-based secure attacks on GPUs.

As described in Section II-B, BCoal implements secure GPU architecture for AES by restricting possible memory coalescing levels of a single warp. Since a memory coalescer in a GPU load/store unit generates up to 16 CMRs when a warp accesses T-tables of AES, the memory coalescer of BCoal always creates 16 memory transactions if the number of CMRs is in the range from 2 to 16. Note that BCoal fully enables memory coalescing if the memory transactions from a warp highly coalesce (i.e. the number of CMRs is one). Since the number of memory transactions that access T-tables is always 16 regardless of encryption keys, BCoal can hide the differences in AES encryption latency influenced by the keys. Note that the correlation-based attacks exploit the correlations between *AES execution time* and the number of CMRs computed using estimated keys. Hence, BCoal can hide the encryption keys that would exhibit the strongest correlation on non-secure GPUs.

However, BCoal results in significant performance drops since it creates redundant memory transactions to make the number of CMRs equal to the upper bound of an original CMR count range. In this paper, we call these redundant memory transactions created for hiding the regular memory coalescing rule as *dummy requests*. Dummy requests access the GPU memory subsystem like other normal memory transactions, however, the data fetched by the dummy requests are not written to register files. As mentioned previously, BCoal always generates 16 CMRs if the original CMR count

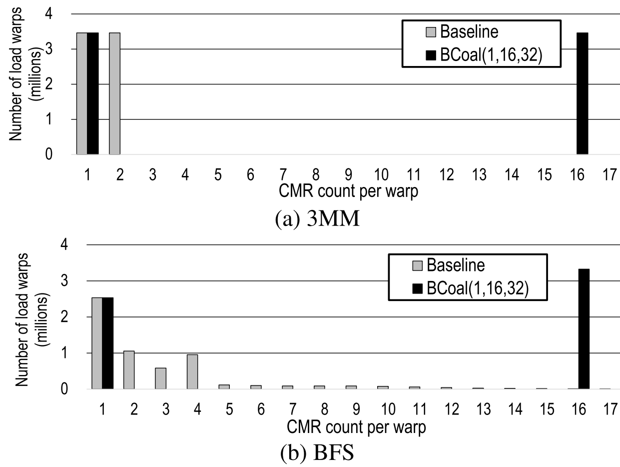

FIGURE 3. Warp counts by the number of CMRs per warp.

is in the range from 2 to 16. Namely, BCoal added 4 dummy requests to the original memory transactions from the memory coalescer if the original CMR count in a warp is 12. Since BCoal's modified memory coalescing rules are applied to all warps which generate CMRs in the same CMR count ranges, dummy requests are added to the normal warps which do not require secure executions. Such unnecessary dummy requests also consume resources in GPU memory subsystem, thus GPU performance is downgraded.

We analyze why the unnecessary dummy requests for secure GPU architecture result in performance drops. Figure 3 shows the counts of load warps sorted by the number of CMRs generated from a single warp for 3MM [19] and BFS [20] applications. We collected the data exhibited in Figure 3 using GPGPU-Sim that configures NVIDIA GTX 480 as listed in Table 2. Baseline is a normal insecure GPU architecture and BCoal(1, 16, 32) represents the secure GPU architecture that implements the defense against the correlation-based attacks. The numbers within the brackets represent the available memory coalescing levels restricted by BCoal. Namely, BCoal(1, 16, 32) creates 1 memory request if the original CMR count is 1 (i.e. perfectly coalesced), 16 memory requests for the original CMR count range from 2 to 16, and 32 requests for the range from 17 to 32. As shown in the figure, load warps of 3MM generate only 1 and 2 CMRs per warp on the baseline GPU. However, BCoal added 14 dummy requests to generate 16 CMRs if the original CMR count from a warp is 2. Thus for 3MM, the total number of CMRs increases by 5.67 times, and execution time is also increased by 3.67 times. For BFS the total number of CMRs increases by 3.36 times compared to the baseline. Table 1 summarizes the total number of CMRs and the execution time of 3MM and BFS when BCoal (1, 16, 32) is employed. All performance data are normalized to the performance metrics on the baseline GPU.

Our motivation study exhibits that the dummy requests augmented for hiding the relation between CMRs and encryption keys result in significant performance drops for general-purpose applications. It is because the prior solution creates lots of dummy requests to fill the predefined quotas (i.e. the possible coalescing levels by BCoal) as shown in Figure 3. A possible solution that can reduce the number of generated dummy requests is allowing more memory coalescing levels. For instance, BCoal can create only 2 more dummy requests if one of the possible coalescing levels is set as 4 and the original number of CMRs is 2. We study the performance changes by permitted coalescing levels as exhibited in Figure 4. We measure the total number of CMRs and the execution time of 3MM and BFS by changing the permitted coalescing levels of BCoal. Our analysis results reveal that the number of CMRs decreases and the performance overhead by BCoal is obviously mitigated as more memory coalescing levels are allowed. Note that the performance by BCoal is slightly lower than the baseline GPU when 9 memory coalescing levels are allowed. BCoal that allows 9 levels of memory coalescing exhibits 3.40 times and 1.82 times of performance uplifts for 3MM and BFS respectively compared to BCoal(1, 16, 32). However, this approach cannot realize secure GPU architecture against the correlation-based attacks. It is because attackers could detect correlations between execution times and the permitted coalescing levels affected by encryption

TABLE 1. CMR counts and execution time by BCoal.

Apps	CMR count	Execution time
3MM	5.67×	3.67×
BFS	3.36×	1.80×

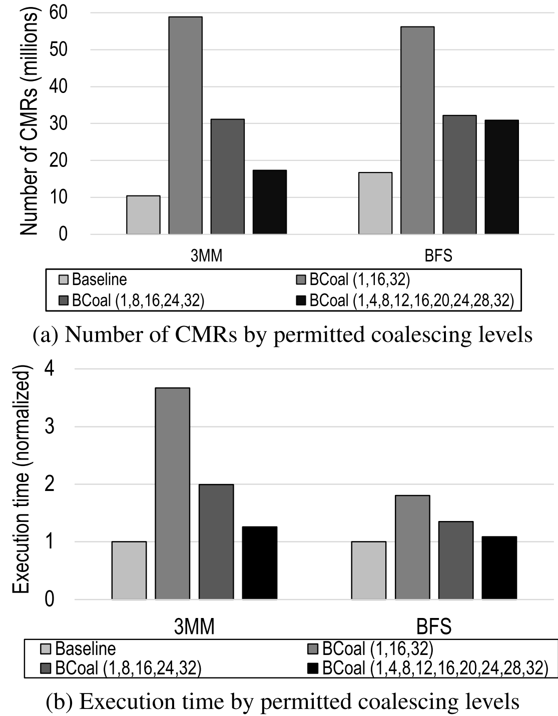

FIGURE 4. Performance changes by allowed coalescing levels.

```
__global__ void AES_encrypt(const uint *pt,uint *rek,…){
uint tid = blockIdx.x * blockDim.x + threadIdx.x;

uint offset = (NUM_THREADS * NUM_BLOCKS + tid) << 2;

s0 = pt[offset + 0] ^ rek[0];
...
t0 = Te0[s0 >> 24] ^ ...;
...
}
```

(a) Deterministic loads (pt[])

```
__global__ void AES_encrypt(const uint *pt,uint *rek,…){
uint tid = blockIdx.x * blockDim.x + threadIdx.x;
uint offset = (NUM_THREADS * NUM_BLOCKS + tid) << 2;
s0 = pt[offset + 0] ^ rek[0];
...
t0 = Te0[s0 >> 24] ^ ...;
...
}
```

(b) Non-deterministic loads (Te0[])

FIGURE 5. Different load types in AES.

keys although more samples are required to leak the secret keys.

IV. GhostLeg

As investigated in the previous section, the prior architectural defense approaches for realizing secure GPU architecture exhibit significant performance overhead. In order to reduce the performance burdens for secure GPU architecture, we propose *GhostLeg* – an efficient architectural defense solution against the correlation-based attacks on GPU. As motivated in Section III the performance overhead by the prior solutions is caused by unnecessary dummy requests augmented to the warps which do not require secure executions. GhostLeg classifies load instructions into two groups - the load warps that generate a fixed number of CMRs and the load warps that create CMRs based on the indexes computed using user data or secret data. Our analysis of the AES kernel reveals that the correlation-based attacks exploit the load warps that generate CMRs affected by AES encryption keys (i.e. non-fixed data). Then GhostLeg generates dummy memory requests only for the warps whose memory coalescing levels need to be hidden. GhostLeg can efficiently pinpoint the load warps that need secure executions by propagating secure flags whenever register operands are collected for executions. Furthermore, GhostLeg creates dummy requests which exhibit similar data fetch latencies compared to the original CMRs. In addition, we also propose *GhostLeg-Key* which applies secure executions only for loads whose indexes are generated from specified *secure data*.

A. CLASSIFICATION OF LOADS

In order to reveal the characteristics of the critical loads that are exploited by the correlation-based attacks, we analyze the AES code ported to a GPU domain. Figure 5 shows a snippet of the AES CUDA code that includes two different types of loads. Note that each thread from a GPU kernel usually accesses arrays using thread and/or block ids assigned to the thread. It is because GPU's execution model favors parallel computations using array elements. In the code of Figure 5a the index of array `pt` is computed from the thread id (`tid`) assigned to each thread. The memory transactions from the load warp that accesses `pt` highly coalesce since threads within the same warp exhibit a fixed stride between indexes of adjacent threads. The load warps accessing `pt` usually generate the same number of CMRs since indexes are computed using constant parameters and thread/block ids. Note that the correlation-based attacks exploit the proportionality in kernel execution times and the number of generated CMRs. Thus these types of loads that generate the fixed number of CMRs per warp are immune to the correlation-based attacks. In the prior article, such types of loads are called *deterministic* loads (D-loads) since data array indexes are calculated only using fixed values (i.e. parameters and thread/block ids) [21]. In this paper, we will also use the same name for such types of loads.

On the other hands, the index of T-table `Te0` is computed using a plain text data (`pt`) and an encryption key (`rek`) as shown in Figure 5b. Unlike the deterministic loads, the indexes of `Te0` are changed by values of the plain text data and the encryption key. Thus the number of CMRs generated from this load warp varies by the encryption key obviously. Since the correlation-based attacks exploit the CMR counts affected by the encryption keys, these types of loads are critical attack surfaces. We call these types of loads as *non-deterministic* loads (ND-loads) since the indexes of such loads are computed using user variables. Note that attackers can calculate the number of CMRs that access the T-tables in the last round of AES using estimated keys as described in Section II-B. The load warps that access the T-tables of the last round are also ND-loads since the indexes of the loads originate from the plain text data and the encryption keys. Hence, GhostLeg selectively applies the architectural defense schemes only to *ND-loads* to minimize performance overhead.

Although GhostLeg can effectively classify D-loads and ND-loads from load warps to hide memory coalescing for ND-loads only, GhostLeg may create unnecessary dummy requests for the general applications that include many ND-loads. For instance, more than 50% of load warps are ND-load warps for BFS. In that case, GhostLeg will apply the defense schemes to ND-loads in BFS, thus the performance of BFS will go down due to increased memory transactions. However, BFS does not require secure executions on GPU. In order to avoid the meaningless performance drops by unnecessary dummy requests, GhosLeg needs to create the dummy requests only for the ND-loads that handle *secret* data. In the example of the above AES code, the encryption keys (`rek[]`) are secret data that demand secure executions. GhostLeg can further filter out unnecessary ND-loads from secret executions if the secret data flags specified by users or compilers are delivered to GPU and the defense schemes are only applied to the ND-loads originating from the secret

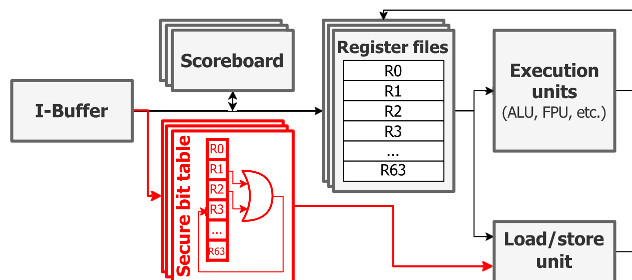

FIGURE 6. Secure bit generation for identifying secure loads.

data. Programmers can specify memory spaces for user data such as *local* and *shared* spaces in CUDA. Data fetched from the variables specified by such declaration specifiers are translated into load instructions of parallel thread execution (PTX) directives like *ld.local* and *ld.shared*. Likewise, GhostLeg can receive a PTX directive that specifies *secret* data to pinpoint the ND-loads whose indexes are computed using the secret data. In this paper, we call this scheme *GhostLeg-Key*. We will describe how GhostLeg detects ND-loads or ND-loads originating from secret data later.

B. IDENTIFYING SECURE LOAD WARPS

The main idea of GhostLeg is to disable memory coalescing only for *secure* load warps which require secure executions. As mentioned in Section II, the correlation-based attacks exploit the observation that the number of CMRs generated from a load warp varies by secret user data. In order to conceal such relation while minimizing performance overhead, GhostLeg selectively disables the memory coalescing for the load warps whose indexes are determined by secret data. Namely, GhostLeg first selects the load warps that possibly need secure executions, then GhostLeg generates dummy requests to hide the diverged memory coalescing levels by the secret data. Now we will describe how this mechanism of GhostLeg is implemented on the baseline non-secure GPU architecture.

Figure 6 depicts the architectural implementation of the GhostLeg mechanism that identifies *secure* load warps. In the figure, we represent the hardware unit and the data flows newly added for GhostLeg as red-colored lines. GhostLeg is implemented on the execution path in an SM as shown in the figure. Note that a load warp requires a secure execution if the addresses of the load warp are computed using secret data. Thus GhostLeg checks if the source registers of a load warp originate from secret data. Since the addresses of a load warp can be computed using sequences of multiple instructions, GhostLeg tracks the propagation of register data using *secure* bits assigned to registers in a warp. Note that each SM employs a scoreboard per warp to check the readiness of source and destination registers when an instruction is at the top of the instruction fetch queue of each warp. GhostLeg augments a *secure bit* per register in the warp scoreboard to indicate whether the corresponding register holds secure data or not. As shown in Figure 6, the secure bit of a destination register is updated as the result of *OR* operation from source registers in a warp instruction. Namely, if one of the source registers holds secure data, the secure bit of the source register is propagated to a destination register. Such secure bit updates do not affect the execution paths of an SM since the secure bits can be updated in parallel while the scoreboard is accessed. When a load warp is issued to a load/store unit in an SM, GhostLeg checks the secure bit of the source register that is used for computing memory addresses. If the secure bit is valid, GhostLeg sends *secure load* signal to the load/store unit to notify the current load warp requires secure executions.

When a warp is initiated in an SM, the secure bits of all registers assigned to the warp are *reset* because the registers do not hold any valid data initially. Thus GhostLeg set the secure bits when data are newly allocated to registers. GhostLeg employs two different policies, *GhostLeg-ND* and *GhostLeg-Key*, for setting up the secure bits. We will describe these two policies.

1) GhostLeg-ND

As mentioned in Section IV-A, the correlation-based attacks exploit the various coalescing levels in ND-loads. In order to guarantee secure executions, GhostLeg-ND regards all ND-loads as *secure* loads. In order to detect ND-loads GhostLeg-ND set the secure bit of a register as *valid* if the corresponding register is updated by the load instructions that access *global*, *local*, and *shared* memory spaces. Note that programmers can specify data spaces of variables using CUDA primitives [22]. The corresponding PTX directives (e.g. *.global*, *.local*, and *.shared*) are included to load instructions based on the memory spaces of user data when CUDA codes are compiled. Since AES secret keys are user-specific data, the key data can be declared using one of those memory space primitives. Once GhostLeg-ND sets the secure bits of the destination registers in such load instructions, the secret bits are propagated to other registers as illustrated in Figure 6. GhostLeg-ND does not require any modifications in the CUDA codes since GhostLeg-ND automatically detects ND-loads using the existing PTX primitives.

2) GhostLeg-Key

For setting up the secret bits of the registers updated by data fetch instructions, GhostLeg-Key relies on a specifier that indicates secret data to be protected. Unlike GhostLeg-ND that sets secret bits when general load warps are issued, GhostLeg-Key only sets the secret bits if the destination register is written by the load instructions specified by a special PTX directive. When GhostLeg-Key is employed, programmers can specify secret data that require secure executions using a special primitive. Then when the secret data are fetched, a CUDA compiler adds a PTX directive *.secret* that follows a general load instruction like *ld.global.secret*. Namely, the secret bit is set if the corresponding register is a destination register of a load instruction specified by *.secret*. GhostLeg-Key can minimize the performance overhead for

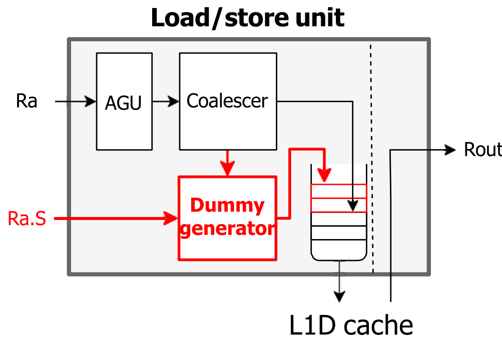

FIGURE 7. Modified load/store unit for GhostLeg.

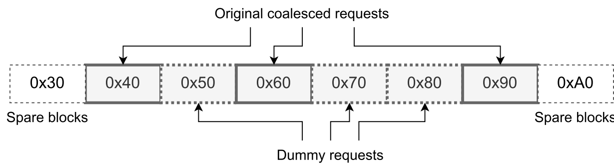

FIGURE 8. Mechanism of dummy request generation.

secure executions since GhostLeg-Key pinpoints the load warps whose memory coalescing behavior is influenced by secret data.

C. GENERATING DUMMY REQUESTS

GhostLeg creates dummy requests to conceal the memory coalescing levels determined by secret data. Figure 7 exhibits the modified load/store unit that supports generating dummy memory transactions by GhostLeg. The hardware units newly added for GhostLeg are represented using a red color. GhostLeg reads the secure bit of the source register that is used for computing addresses of a load warp. If this secure bit is valid, GhostLeg issues the *secure load* signal to the load/store unit. In the figure, `Ra` is the source register of a load warp and `Ra.S` means the secure bit of `Ra`. If `Ra.S` is valid, the dummy request generator in the load/store unit is activated. Then dummy memory transactions are enqueued into the request queue along with the original memory requests. GhostLeg adds the dummy requests until the total number of memory requests (i.e. original + dummy requests) reaches the pre-defined number. In order to hide the deterministic behavior of the memory coalescing, we can set the pre-defined threshold level to be the maximum number of CMRs possibly generated from secure load warps. For instance, the maximum number of CMRs generated from the load warps that access T-tables is 16 in AES. Thus it is enough to conceal memory coalescing behaviors if we set the threshold level to 16. This policy is similar to BCoal's bucketing-based memory coalescing, which adjusts the bucketing levels to hide the memory coalescing behaviors observed in AES [14].

In order to guarantee secure executions, the characteristics (e.g. data fetch latency) of the dummy requests should be similar to the original memory transactions. Otherwise, attackers may observe correlations between the execution times and the number of original CMRs by increasing the sample count even though the execution times are scrambled by augmented dummy requests. We can expect that all threads within a warp exhibit similar data fetch latencies if all the threads access the elements of the same data array. Thus dummy requests can exhibit similar latencies if the dummy transactions request other elements of the same data array accessed by the original transactions. However, GPU hardware is not aware of the ranges of data arrays basically, thus allocating the dummy request addresses based on the range of a data array is not a practical approach.

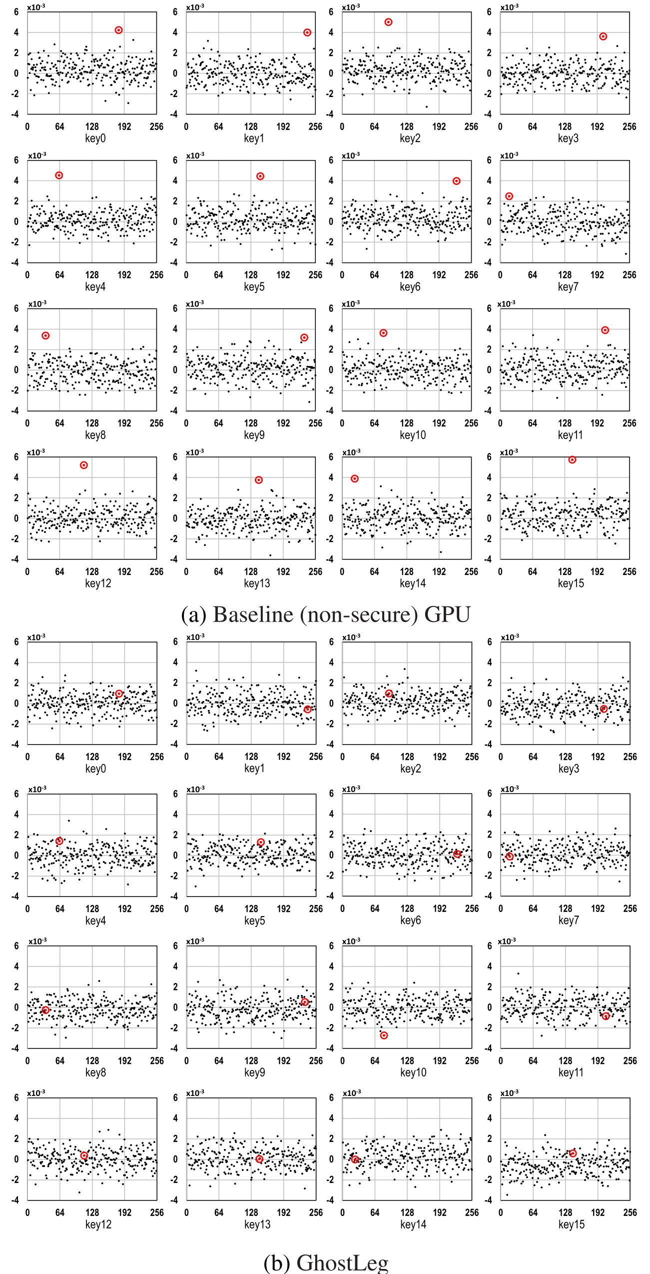

FIGURE 9. Correlations by estimated keys.

TABLE 2. GPU configurations.

Parameter	Fermi (GTX480)	Pascal (Titan X)
SMs	15 SMs @ 1.4GHz	28 SMs @ 1.417GHz
Threads / warp	32	32
Warps / SM	48	64
CTAs / SM	8	16
Scheduler	GTO, 2 per SM	GTO, 4 per SM
Register file	128KB per SM	256KB per SM
Coalescing width	64B	64B
L1D cache	16KB per SM, 4-way	24KB per SM, 4-way
L2 cache	786KB, 8-way	3MB, 16-way
DRAM	GDDR5 @ 924MHz	GDDR5X @ 2.5GHz

Figure 8 illustrates how GhostLeg decides the addresses of dummy requests to guarantee secure executions. Let us assume that the size of the memory coalescing block is 16 bytes. In this example, `0x40`, `0x60`, and `0x90` are access addresses by original requests generated from the memory coalescing logic. GhostLeg first computes the address range (i.e. `0x40` – `0x90`) of the original transactions created by a load warp. Then GhostLeg searches the missing blocks (i.e. `0x50`, `0x70`, and `0x80`) within the range to allocate these addresses for dummy requests. Since these blocks are within the range of the original transactions in a warp, the data accessed by the dummy requests are included in the same data array. If GhostLeg cannot create enough number of dummy request within the original transaction range, GhostLeg uses the adjacent block addresses (i.e. `0x30` and `0xA0` in Figure 8) for additional dummy requests. Note that the memory transactions that access the adjacent blocks can exhibit similar characteristics considering data locality. Consequently, GhostLeg effectively conceals the execution time changes by the number of original CMRs since GhostLeg augments the homogeneous memory transactions as dummy requests.

The dummy request generation by GhostLeg does not change the timing of the original CMRs. As shown in Figure 7 the dummy request generator receives minimal information such as the address range of the original CMRs, then the dummy request generator runs simultaneously along with the memory coalescer. The generated dummy requests are sent to the cache request queue after the original CMRs are enqueued. Consequently, GhostLeg does not affect the existing pipelines in the load/store unit.

V. EVALUATION

We evaluate GhostLeg with cycle-accurate GPU simulator, GPGPU-Sim V3.2.2 [23]. We implement two different policies, *GhostLeg-ND* and *GhostLeg-Key* as described in Section IV. We configure GPGPU-Sim with Fermi and Pascal architectures as listed in Table 2.

Workloads: In order to evaluate the performance overhead by GhostLeg and other architectural defense approaches, we run the general-purpose GPU applications as listed in Table 3. We categorize the workloads into memory-intensive (MI), memory-moderate (MM), and compute-intensive (CI) applications based on the fractions of load warps out of all

TABLE 3. Workloads.

App	Description	Ratio of loads	Ratio of ND-loads
Memory-intensive (MI)			
3MM	3D matrix multiplications [19]	22.16%	0.00%
DIT	Multi-resolution analysis [19]	21.97%	0.00%
SCG	Stream cluster [20]	17.98%	4.16%
BTR	B+-tree [20]	16.21%	62.11%
HTW	Heartwall tracking [20]	11.14%	0.55%
BFI	Breadth first search [23]	10.67%	57.95%
SM	String match [24]	10.27%	96.62%
Memory-moderate (MM)			
JC2	2D Jacobi stencil compute [19]	9.57%	0.00%
BFF	Breadth first search [25]	8.44%	20.27%
HIS	Saturating histogram [25]	7.79%	0.02%
LBM	Lattice-Boltzman method [25]	7.79%	0.00%
SR1	SRAD V1 [20]	7.49%	34.30%
PFF	Particle filter [20]	7.44%	31.22%
LU	LU decomposition [19]	7.32%	0.00%
BFS	Breadth first search [20]	7.17%	53.51%
GAF	Gaussian elimination [20]	6.09%	0.00%
Compute-intensive (CI)			
LPS	3D Laplace solver [23]	1.89%	0.00%
MUM	MUMmerGPU [23]	1.79%	99.52%
PTH	Pathfinder [20]	1.75%	0.00%
SGE	SGEMM [25]	1.55%	0.00%
MRG	MRI-gridding [25]	1.43%	21.61%
HSP	3D hotspot [20]	0.65%	0.00%
CUT	Coulombic potential [25]	0.21%	100.00%

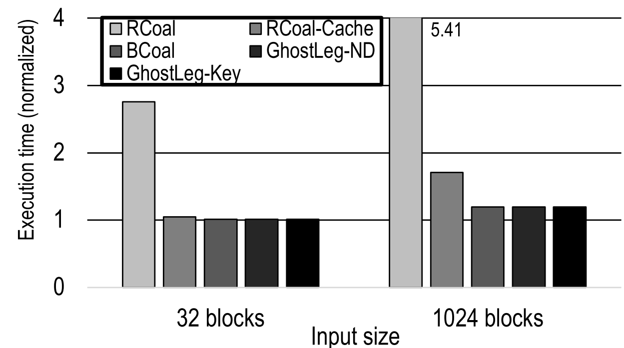

FIGURE 10. Normalized execution time of AES on Fermi (GTX 480).

warp instructions. MI applications include more than 10% of load warps, and MM applications have 10% – 5% of load warp instructions. For CI applications the fraction of load warps is less than 5%. We sort the workloads by the fraction of load warps. We also exhibit the fraction of ND-loads to all load warps in the last column of the table.

A. DEFENSE AGAINST THE ATTACKS

In order to evaluate the effectiveness of our defense mechanism, we study the correlation-based attacks that target AES encryption keys. We use the AES kernel from OpenSSL 0.9.7 library ported to GPU domains. To extract the AES encryption keys, We apply the attack approaches presented in [12]. As introduced in Section II-B, we measure the correlations between the execution times of the AES kernel and the number of CMRs by the estimated encryption keys. Figure 9 shows the correlations by the estimated key values

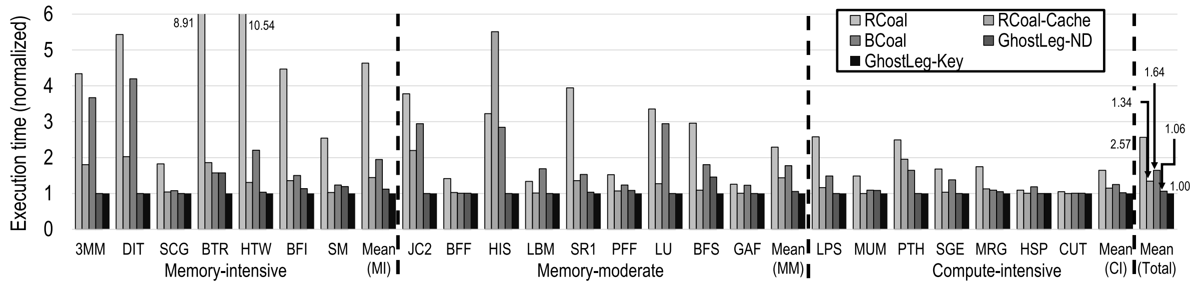

FIGURE 11. Normalized execution time on Fermi (GTX 480).

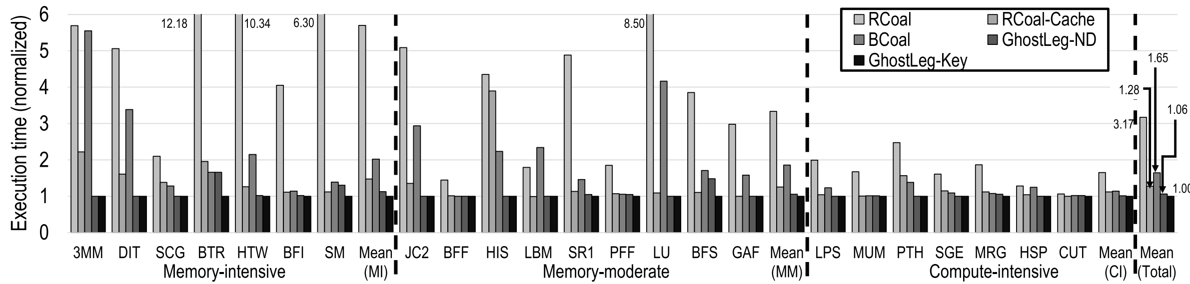

FIGURE 12. Normalized execution time on Pascal (Titan X).

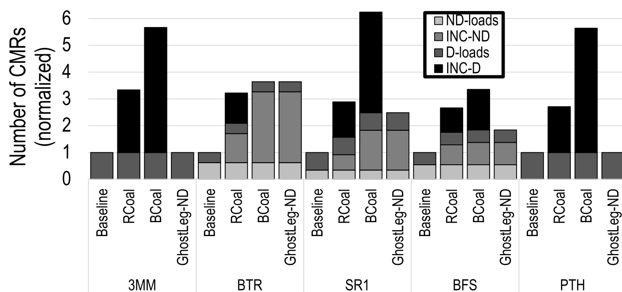

FIGURE 13. Fractions of CMRs by load warp classes.

of all 16 bytes (128 bits) when the correlation-based attacks are fulfilled on the non-secure baseline GPU and GhostLeg for 1 million input samples. As shown in Figure 9a we can observe the real encryption key (a red dot) exhibits the highest correlation among 256 estimated key values (0–255) for every key byte. Note that attackers can clearly identify the encryption keys when the AES kernel runs on the non-secure GPU. On the other hand, we can observe the correlations by the same estimated key values are low when GhostLeg is employed. We cannot recognize any differences in correlations among the true encryption keys and falsely estimated keys as shown in Figure 9b. Namely, attackers cannot figure out the encryption keys since GhostLeg conceals the memory coalescing for load warps that access T-tables using the secret keys. Consequently, our evaluation result exhibits that GhostLeg can effectively protect AES encryption keys from the correlation-based attacks.

B. PERFORMANCE OVERHEAD

As mentioned in Section III, the severe performance overhead by the prior architectural defense approaches is a critical problem in implementing secure GPU architecture. As shown in Figure 10, we first evaluate the performance overhead by GhostLeg for AES since the correlation-based attacks target AES ported to a GPU domain. We measure the execution time of the AES kernel that encrypts 32 blocks and 1024 blocks of plain text. AES that handles 32 blocks creates only one warp (i.e 32 threads), thus we can study the execution time of a single warp by defense approaches. On the other hand, AES on GPU usually processes large plain text data to exploit massive thread-level parallelism of GPU. Thus when the number of plain text blocks is large, AES creates many warps to be assigned across multiple SMs. In order to compare the performance of GhostLeg, we implement RCoal and BCoal as presented in [13] and [14]. RCoal shuffles threads assigned to subwarps to randomize memory coalescing behaviors [13]. Note that RCoal affords stronger defense capability however performance drops more significantly as the number of subwarps increases. For RCoal we set the number of subwarps as 4, thus RCoal can provide reasonable security levels with low performance overhead. RCoal only allows data to be fetched from DRAM to guarantee secure executions thus it exhibits severe performance drops. RCoal-Cache exploits cache hierarchy in GPU to improve performance over original RCoal, however, it cannot guarantee secure executions due to merged transactions in MSHRs [14]. BCoal implements the bucketing-based memory coalescing. We configure the bucket sizes of BCoal as (1, 16, 32) since BCoal can support the secure executions for AES with low performance overhead using this configuration [14]. We implement GhostLeg-ND and GhostLeg-Key as described in Section IV-B.

We also evaluate the performance overhead by secure architecture solutions for general-purpose applications listed in Table 3. We run the benchmarks until the number of

committed instructions reaches 1 billion or an application run completes. The execution time of each benchmark by the secure GPU solutions is normalized by the execution time on the baseline GPU. We measure the performance on Fermi (in Figure 11) and Pascal (in Figure 12) architecture models. As shown in the figures, GhostLeg exhibits lower performance overhead compared to the prior defense solutions. For both Fermi and Pascal architectures, the average performance overhead by GhostLeg-ND is only 6%. The performance of GhostLeg-ND is 142% and 54.7% higher than RCoal and BCoal respectively on Fermi. On the modern architecture (Pascal) GhostLeg-ND is more efficient as the performance is higher than RCoal and BCoal by 199% and 55.7% respectively. It is because GhostLeg-ND selectively applies the memory coalescing only for D-loads. For some memory-intensive applications such as BTR, BFI, and SM that include a large fraction of ND-loads, the performance can be degraded by GhostLeg-ND since secure executions are applied for ND-loads. However, GhostLeg-ND exhibits better performance for such applications compared to the prior defense solutions. The performance by GhostLeg-Key is equivalent to the performance of the baseline machine since we don't need to specify *secret* data for the general-purpose applications that do not require secure executions. Consequently, our performance evaluation results reveal that GhostLeg is an efficient secure GPU architecture solution that can protect secret data from the correlation-based attacks.

C. TRAFFIC ON MEMORY SUBSYSTEM

In order to figure out the sources of performance drops by the secure GPU architecture solutions, we study the memory traffic by D-loads and ND-loads for several applications. Note that as mentioned in Section III the defense solutions nullify the memory coalescing, which is an essential part of improving GPU performance. Figure 13 presents the breakdown of CMRs by load warp classes for selected benchmarks. In the bar graphs, *ND-loads* represents the fraction of CMRs generated from ND-load warps, which is equivalent to the number of CMRs from ND-load warps divided by the total number of load requests on the baseline GPU. *INC-ND* means the number of *increased* CMRs from ND-load warps by the defense approaches. All these numbers are normalized to the total number of load requests on the baseline machine. In the same way, *INC-D* represents the fraction of the increased CMRs from D-load warps due to the defense solutions.

As shown in Figure 13, all CMRs in 3MM are generated from ND-loads, thus GhostLeg-ND does not increase load transactions. However we observe a large number of dummy requests are created from D-loads by RCoal and BCoal, thus these solutions exhibit significant performance overhead for 3MM. For BTR we observe all defense approaches exhibit more than 3× of load requests compare to the baseline. GhostLeg-ND also creates many dummy requests since BTR includes more than 60% of ND-loads, thus GhostLeg conceals the memory coalescing for these ND-loads. For BTR the performance by GhostLeg-ND is similar to BCoal and 467.3% higher than RCoal-Cache. Since GhostLeg-ND selectively enables secure executions only for ND-loads, GhostLeg-ND usually exhibits lower traffic by load requests compared to RCoal and BCoal. Again, GhostLeg-Key does not increase the number of memory transactions if programmers do not specify *secret* data.

D. ENERGY CONSUMPTION

Figure 14 shows the energy consumption by the defense solutions. We use GPUWattch to measure the energy consumption while running the applications on GPGPU-Sim [26]. All energy consumption data are normalized to the energy consumption of the baseline GPU. The energy consumption increased by GhostLeg is only 5.62%, which is much lower than other defense approaches. Even though GhostLeg-ND increases memory traffic slightly, the average power consumed by GhostLeg-ND is similar to the baseline. Our evaluation exhibits that GhostLeg is an energy-efficient approach.

E. AREA OVERHEAD

GhostLeg can implement the architectural defense against the correlation-based attacks with minimal hardware resources. In order to estimate the area of the additional hardware resources for GhostLeg, we design the RTL models of GhostLeg's hardware components. As mentioned in Section IV-B, GhostLeg manages the secure bit tables to identify secure load warps. Note that GhostLeg stores a single secure bit per register in a warp to track the sources of load addresses. Thus GhostLeg requires 64 secure bits per warp for Fermi architecture since Fermi supports up to 64 registers per warp. As the maximum number of concurrent warps per SM is 48 for Fermi, GhostLeg includes 3,072 secure bits per SM for implementing the secure bit tables. In addition, GhostLeg implements the dummy request generator that works along with the memory coalescer in the load/store unit as described in Section IV-C. As Fermi includes one load/store unit per SM, the area of one dummy request generator is augmented per SM by GhostLeg.

In order to estimate the area overhead by GhostLeg, we synthesize the RTL models of the security bit tables and the dummy request generator using 45nm FreePDK library [27]. The estimated area of a single secure bit table is 1655.3 μm^2, thus for Fermi architecture 48 secure bit tables occupy 79,454.4 μm^2. The synthesis tool reports the estimated area occupied by a dummy request generator is 947.6 μm^2. For Fermi architecture, GhostLeg increases the area of an SM by 0.36% considering the area of a single SM in GF100 is estimated to be 22 mm^2 [28].

VI. RELATED WORK

A. SECURITY ATTACKS ON GPU

Security research is a well-explored area in the CPU domain. Many researchers have revealed security attacks that exploit the vulnerabilities in CPU microarchitecture and side/covert

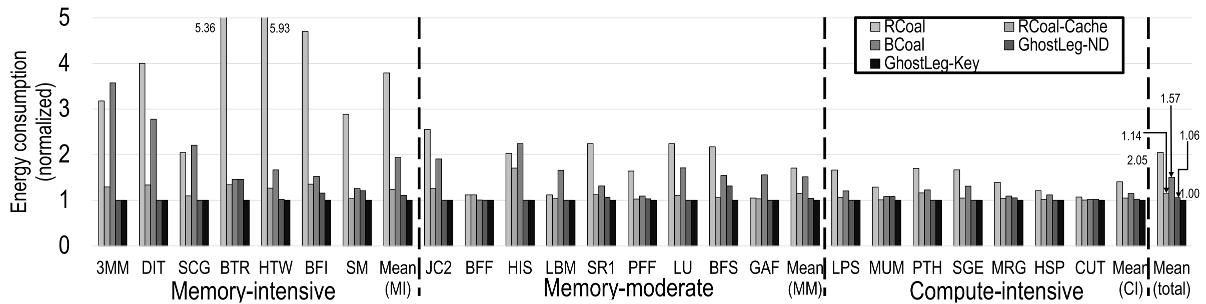

FIGURE 14. Normalized energy consumption on Fermi (GTX 480).

channels in memory hierarchy [1], [2], [3], [4], [5], [6], [7], [9], [29], [30], [31], [32], [33]. As more general-purpose applications including applications in a security domain have been ported to a GPU domain, research on secure GPU architecture is getting more critical. Since GPU employs its unique single-instruction multiple-thread (SIMT) architecture and software execution models to support hundreds of concurrent threads, researchers have disclosed several security attacks that exploit the unique architectural vulnerabilities found in GPU.

Several studies revealed security attacks can exploit the timing side-channels of GPU. Such attacks exploit the observations that the execution time of the popular security kernels executed on GPU can vary by secret data. Thus attackers can figure out the secret data that exhibits the highest correlation with the large set of execution latencies of the security kernels. Jiang et al. presented that encryption keys of AES can be leaked using correlations between the execution times of AES and the number of memory requests created from a single warp for estimated encryption keys [12]. The same authors also disclosed attackers can leak a secret key of AES exploiting the different timings caused by bank conflicts in GPU's shared memory [34]. Ahn et al. revealed the characteristics of local caches in the modern GPU architectures are different from the original cache design. They disclosed a new correlation-based attack that uses negative correlations and cache collisions [35]. Luo et al. presented that encryption keys of RSA ported to a GPU domain can be extracted by measuring the execution latencies of several window units of RSA [36].

Some studies presented covert channel-based attack models that exploit the contentions on interconnection networks inside of a GPU or communication links across multiple GPUs. Ahn et al. analyzed interconnection network organizations of a modern GPU architecture to reveal the hierarchical structure of the interconnection network [37]. They disclosed attackers can design contention-based attack models since multiple SMs in the same cluster share the resource of the hierarchical interconnection structures. Dutta et al. presented that a spy process in one GPU can cause contention on L2 caches in other discrete GPUs connected via multi-GPU communication links [38]. By exploiting the inter-GPU communications, they designed prime+probe type attacks across multiple GPUs. Naghibijouybari et al. disclosed that keystrokes and neural network models can be leaked by a spy process that accesses performance counters [39]. Consequently, researchers have disclosed that security attack models can be designed by exploiting vulnerabilities in GPU hardware/software.

B. DEFENSES AGAINST GPU SECURITY ATTACKS

Researchers proposed defense mechanisms against the correlation-based attacks on GPU. Such defense approaches tackle the relation between AES encryption keys and the execution time of AES since the number of memory requests varies by the encryption keys. Kadam et al. proposed RCoal which randomizes the threads assigned to subwarps [13]. Thus the number of memory transactions generated by memory coalescing is decided rather randomly when RCoal is employed. The researchers also proposed a defense approach that restricts the possible coalescing levels to conceal the deterministic behaviors of the memory coalescing [14]. Karimi et al. proposed a hardware-based obfuscating mechanism that changes memory coalescing width and a software-based approach that permutes mapping table structures [40]. Lin et al presented a software-based modification that changes the compositions of T-tables to make AES generates a fixed number of memory requests [41]. Ahn et al. proposed a defense mechanism, called Trident, which makes memory transactions bypass the L1 cache randomly [35]. Even though these defense solutions can be effective to protect secret data from the correlation-based attacks, such approaches cause significant performance drops.

VII. CONCLUSION

In this paper, we propose GhostLeg, an efficient architectural defense mechanism against correlation-based GPU security attacks. GhostLeg tackles significant performance overhead induced by the existing secure GPU architectures. By analyzing the AES kernel ported to a GPU domain, we reveal only non-deterministic loads whose indexes are computed from secret data are vulnerable to the correlation-based attacks. GhostLeg first sets a secure bit when a register is updated by user data (*GhostLeg-ND*) or secret data specified by programmers (*GhostLeg-Key*). Then the secret bits assigned to registers are propagated until the registers are used for

computing addresses of a load warp. When a load warp is issued, GhostLeg checks the secret bit of the source register to enable secure executions if the secret bit is valid. Hence, GhostLeg filters out unnecessary secure executions for load warps to minimize the performance overhead by augmented dummy requests. Our evaluation results show that GhostLeg-ND results in only 6.29% of performance overhead. GhostLeg-ND exhibits 54.7% higher performance compared to the state-of-the-art defense solution.

ACKNOWLEDGMENT

(Jongmin Lee and Seungho Jung contributed equally to this work.)

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, ''Spectre attacks: Exploiting speculative execution,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, Jun. 2019, pp. 1–19.

[2] V. Kiriansky and C. Waldspurger, ''Speculative buffer overflows: Attacks and defenses,'' 2018, *arXiv:1807.03757*.

[3] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, ''Spectre returns! Speculation attacks using the return stack buffer,'' in *Proc. 12th USENIX Conf. Offensive Technol.*, 2018, p. 3.

[4] G. Maisuradze and C. Rossow, ''Ret2spec: Speculative execution using return stack buffers,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2109–2122.

[5] J. Lee and G. Koo, ''Restore buffer overflow attacks: Breaking undo-based defense schemes,'' in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2022, pp. 315–318.

[6] D. A. Osvik, A. Shamir, and E. Tromer, ''Cache attacks and countermeasures: The case of AES,'' in *Topics in Cryptology*. Heidelberg, Germany: Springer, 2006, pp. 1–20.

[7] E. Tromer, D. A. Osvik, and A. Shamir, ''Efficient cache attacks on AES, and countermeasures,'' *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010.

[8] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi. (2010). *Low Voltage Fault Attacks to AES and RSA on General Purpose Processors*. [Online]. Available: http://eprint.iacr.org/2010/130

[9] Y. Yarom and K. Falkner, ''FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack,'' in *Proc. 23rd USENIX Conf. Secur. Symp.*, 2014, pp. 719–732.

[10] Y. Yarom, D. Genkin, and N. Heninger, ''CacheBleed: A timing attack on OpenSSL constant-time RSA,'' in *Cryptographic Hardware and Embedded Systems*. Heidelberg, Germany: Springer, 2016, pp. 346–367.

[11] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley, ''Cache-timing attacks on RSA key generation,'' *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, no. 4, pp. 213–242, 2019.

[12] Z. H. Jiang, Y. Fei, and D. Kaeli, ''A complete key recovery timing attack on a GPU,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 394–405.

[13] G. Kadam, D. Zhang, and A. Jog, ''RCoal: Mitigating GPU timing attack via subwarp-based randomized coalescing techniques,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 156–167.

[14] G. Kadam, D. Zhang, and A. Jog, ''BCoal: Bucketing-based memory coalescing for efficient and secure GPUs,'' in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 570–581.

[15] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, ''CTA-aware prefetching and scheduling for GPU,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 137–148.

[16] J. Daemen and V. Rijmen, *The Design of Rijndael*, vol. 2. Heidelberg, Germany: Springer, 2002.

[17] O. Dunkelman and N. Keller, ''The effects of the omission of last round's MixColumns on AES,'' *Inf. Process. Lett.*, vol. 110, nos. 8–9, pp. 304–308, 2010.

[18] N. Ding and S. Williams, ''An instruction roofline model for GPUs,'' in *Proc. IEEE/ACM Perform. Model., Benchmarking Simul. High Perform. Comput. Syst. (PMBS)*, Nov. 2019, pp. 7–18.

[19] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, ''Auto-tuning a high-level language targeted to GPU codes,'' in *Proc. Innov. Parallel Comput. (InPar)*, 2012, pp. 1–10.

[20] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, ''A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,'' in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Dec. 2010, pp. 1–11.

[21] G. Koo, H. Jeon, and M. Annavaram, ''Revealing critical loads and hidden data locality in GPGPU applications,'' in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2015, pp. 120–129.

[22] NVIDIA. *Parallel Thread Execution ISA*. Accessed: Jun. 17, 2022. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution

[23] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, ''Analyzing CUDA workloads using a detailed GPU simulator,'' in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2009, pp. 163–174.

[24] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, ''Mars: A MapReduce framework on graphics processors,'' in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techn.*, 2008, pp. 260–269.

[25] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, ''Parboil: A revised benchmark suite for scientific and commercial throughput computing,'' *Center Reliable High-Perform. Comput.*, vol. 127, p. 29, Mar. 2012.

[26] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, ''GPUWattch: Enabling energy optimizations in GPGPUs,'' *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, 2013.

[27] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, ''FreePDK: An open-source variation-aware design kit,'' in *Proc. IEEE Int. Conf. Microelectronic Syst. Educ. (MSE)*, Jun. 2007, pp. 173–174.

[28] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, ''Warped-compression: Enabling power efficient GPUs through register compression,'' in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2015, pp. 502–514.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, ''Meltdown: Reading kernel memory from user space,'' *Commun. ACM*, vol. 63, no. 6, pp. 46–56, May 2020.

[30] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, ''RIDL: Rogue in-flight data load,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 88–105.

[31] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, ''CrossTalk: Speculative data leaks across cores are real,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 1852–1867.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, ''Last-level cache side-channel attacks are practical,'' in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.

[33] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, ''Flush+Flush: A fast and stealthy cache attack,'' in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham, Switzerland: Springer, 2016, pp. 279–299.

[34] Z. H. Jiang, Y. Fei, and D. Kaeli, ''A novel side-channel timing attack on GPUs,'' in *Proc. Great Lakes Symp. VLSI*, May 2017, pp. 167–172.

[35] J. Ahn, C. Jin, J. Kim, M. Rhu, Y. Fei, D. Kaeli, and J. Kim, ''Trident: A hybrid correlation-collision GPU cache timing attack for AES key recovery,'' in *Proc. IEEE Int. Symp. High-Performance Comput. Archit. (HPCA)*, Feb. 2021, pp. 332–344.

[36] C. Luo, Y. Fei, and D. Kaeli, ''Side-channel timing attack of RSA on a GPU,'' *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 1–18, Sep. 2019.

[37] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim, ''Network-on-chip microarchitecture-based covert channel in GPUs,'' in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2021, pp. 565–577.

[38] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, ''Spy in the GPU-box: Covert and side channel attacks on multi-GPU systems,'' 2022, *arXiv:2203.15981*.

[39] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, ''Rendered insecure: GPU side channel attacks are practical,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2139–2153.

[40] E. Karimi, Y. Fei, and D. Kaeli, ''Hardware/software obfuscation against timing side-channel attack on a GPU,'' in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Dec. 2020, pp. 122–131.

[41] Z. Lin, U. Mathur, and H. Zhou, ''Scatter- and-gather revisited: High-performance side-channel-resistant AES on GPUs,'' in *Proc. 12th Workshop Gen. Purpose Process. Using GPUs (GPGPU)*, 2019, pp. 2–11.

JONGMIN LEE received the B.S. and M.S. degrees in computer science and engineering from Korea University, Seoul, South Korea, in 2011 and 2013, respectively, where he is currently pursuing the Ph.D. degree in computer science and engineering. From 2016 to 2019, he worked with TmaxSoft, South Korea, as a Researcher. His main research role at TmaxSoft was developing operating system kernels. His research interests include computer architecture and trusted computing.

SEUNGHO JUNG received the B.S. degree in electronic and electrical engineering from Hongik University, in 2020. He is currently pursuing the master's degree in computer science and engineering with Korea University. His research interests include GPU architecture and memory systems. His current research interests include secure processor architecture and supporting secure executions in GPUs.

TAEWEON SUH (Member, IEEE) received the B.S. degree in electrical engineering from Korea University, Seoul, South Korea, in 1993, the M.S. degree in electronics engineering from Seoul National University, in 1995, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2006. He is currently a Professor with the Department of Computer Science and Engineering, Korea University.

YUNHO OH (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea, in 2009, 2011, and 2018, respectively. He is currently working as an Assistant Professor with the School of Electrical Engineering, Korea University. Prior to joining Korea University, he worked as an Assistant Professor at Sungkyunkwan University. From 2019 to 2021, he worked as a Postdoctoral Researcher with the Parallel Systems Architecture Laboratory (PARSA), EPFL, Switzerland. From 2011 to 2014, he worked as a Software Engineer in mobile communications business at Samsung Electronics. His research interests include hardware and software architectures for energy-efficient datacenters, processor architectures (CPUs, GPUs, and neural network accelerators), in-storage processing, memory systems, and high-performance computing.

MYUNG KUK YOON (Member, IEEE) received the B.S. degree in computer engineering and computational mathematics from Washington State University (WSU), Pullman, Washington, USA, in 2011, and the Ph.D. degree in electrical and electronic engineering from Yonsei University, Seoul, South Korea, in 2018. He is currently working as an Assistant Professor with the Department of Computer Science and Engineering, Ewha Womans University. Prior to joining Ewha Womans University, he worked as a Software Developer at Samsung Inc. His research interests include GPU micro-architecture, machine learning accelerators, and parallel programming.

GUNJAE KOO (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, in 2001 and 2003, respectively, and the Ph.D. degree in electrical engineering from the University of Southern California, in 2018. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Korea University. His research interests include computer system architecture and span parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture. Prior to joining Korea University, he was an Assistant Professor with Hongik University. His industry experiences include a Senior Research Engineer with LG Electronics and also a Research Intern with Intel.