LITE: A Low-Cost Practical Inter-Operable GPU TEE

Ardhi Wiratama Baskara Yudha
yudha@knights.ucf.edu
University of Central Florida
Orlando, Florida, USA

Jake Meyer
jmeyer1124@knights.ucf.edu
University of Central Florida
Orlando, Florida, USA

Shougang Yuan
syuan3@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

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

Yan Solihin
Yan.Solihin@ucf.edu
University of Central Florida
Orlando, Florida, USA

Abstract

There is a strong need for GPU trusted execution environments (TEEs) as GPU is increasingly used in the cloud environment. However, current proposals either ignore memory security (i.e., not encrypting memory) or impose a separate memory encryption domain from the host TEE, causing a very substantial slowdown for communicating data from/to the host.

In this paper, we propose a flexible GPU memory encryption design called LITE that relies on software memory encryption aided by small architecture support. LITE's flexibility allows GPU TEE to be co-designed with CPU to create a unified encryption domain. We show that GPU applications can be adapted to the use of LITE encryption APIs without major changes. Through various optimizations, we show that software memory encryption in LITE can produce negligible performance overheads (1.1%) for regular benchmarks and still-acceptable overheads (56%) for irregular benchmarks.

CCS Concepts: • **Computer systems organization** → **Computer architecture**; *Confidentiality*; GPU; • **Memory-Encryption**;

Keywords: GPU TEE, software encryption, memory encryption, GPU enclave

ACM Reference Format:
Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2022. LITE: A Low-Cost Practical Inter-Operable GPU TEE. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524059.3532361

ICS '22, June 28–30, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532361

1 Introduction

Secure and private computation in the cloud is increasingly demanded by cloud computing users. To cater to that, chip manufacturers provide CPU TEE (Trusted Execution Environment), through which the processor provides a root of trust for guaranteeing confidentiality (and sometimes integrity) of computation and data against vulnerabilities in system software. A typical CPU TEE includes key features such as key management, attestation, and *memory security* [14] (memory encryption and/or integrity verification). Note that memory security is far costlier than others due to its continuous application during execution.

GPUs are increasingly widely used in the cloud. However, since current GPUs do not support TEEs, users sacrifice security and privacy when offloading computation to GPUs. Recently researchers have looked at providing TEE on GPUs [7, 23, 26, 27]. While these solutions work, they do not address inter-operability with CPU TEE. The CPU and GPU have their own encryption domains where each is the only one that can decrypt data it encrypted previously. Therefore, for a CPU to send data to a GPU, the data must be decrypted (by the CPU), re-encrypted in software, transmitted to GPU memory, decrypted in software, and then re-encrypted into the GPU encryption domain. Such *encryption domain crossing*, ignored in prior studies, incurs high costs. Figure 1 shows that the crossing overheads contribute to 60% higher execution time and up to 4.2× slowdown (322% overhead), which is much higher than 16% performance overhead from memory security alone. While these numbers were collected across different platforms, it is clear that encryption domain crossing dominates performance concerns.

Achieving inter-operability with CPU requires CPU/GPU TEE co-design, but co-designing is difficult for various reasons, e.g., GPU manufacturers may be different than CPU manufacturers, their design cycles may be different, etc. Furthermore, as an accelerator for the CPUs, the GPU TEE scheme should support a variety of GPU usage scenarios. For example, it may be paired with CPUs that vary in architectures, ISAs, types of CPU TEEs supported, and whether GPUs are paired with a single virtual machine (VM) vs. shared between VMs, etc. Given the wide variety of contexts in which

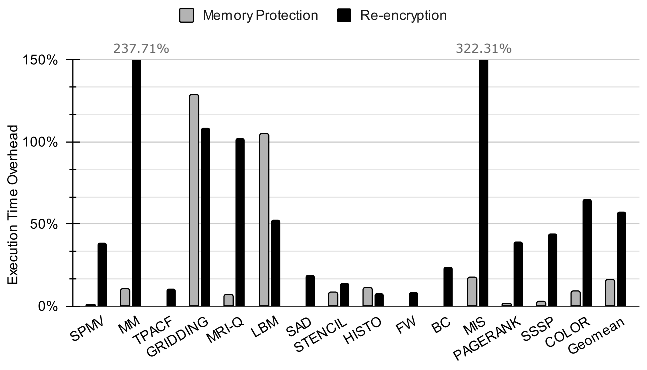

Figure 1. Execution time overheads of memory security with a state-of-the-art scheme (PSSM [27]) and domain crossing re-encryption overheads. The former was collected from GPU simulation with machine configuration from [27], while the latter is from a real machine described in Section 5.

GPUs may be deployed, it is important that GPU TEE support is as flexible as possible.

A flexible GPU TEE design provides additional benefits. Some workloads may process sensitive data, hence confidentiality is required, but confidentiality may not be a priority for many others (e.g., graphics rendering or gaming). Furthermore, even for a single GPU kernel, in some cases, both input and output may be confidential, or only one of either input or output may be confidential.

However, the need for a flexible GPU TEE conflicts with the need to co-design GPU and CPU TEE. Co-designing them requires early coordination of CPU and GPU TEE design, which is difficult to achieve across companies. Furthermore, the wide variety of contexts and workloads in which GPUs could be used is hard to anticipate that early. Building many types of GPU TEEs to provide flexibility incurs a high fixed cost, e.g., supporting a single GPU memory security scheme already requires high die area overheads: one encryption engine for each memory partition, plus metadata caches to keep counters, MACs, and Merkle Tree nodes [27].

To achieve GPU/CPU TEE co-design without sacrificing flexibility, we propose software-based memory encryption, which we refer to as LITE. LITE requires only small hardware support for GPU TEE, and it relegates memory encryption to software. LITE achieves flexibility because software can choose different encryption algorithms to be in accordance with host TEE, selectively choose applications to apply encryption to, and select the subset of data to encrypt in an application, etc. LITE is possible in a GPU because GPU architecture supports explicit data movement (e.g., global to shared memory), unlike a CPU which relies only on caches that implicitly move data. LITE provides a library and APIs that the compiler/programmer can use to keep data encrypted in memory and only decrypt it before use.

To summarize, this paper makes the following contributions:

1. We propose a lightweight GPU TEE (LITE) solution that allows flexible CPU-GPU TEE co-design through the software layer.
2. We present three optimizations, masked shuffle, delayed shuffle, and selective padding, to LITE that significantly improve its performance.
3. We show that despite relying on software for encryption, the optimized LITE incurs low performance overheads, with a geometric mean slowdown of 1.1% for regular applications. However, irregular workloads incur high performance overheads (55.7% on average). This overhead could be reduced by partial encryption to only 10.0% and 44.3% for input-only and output-only encryption, respectively.

The remainder of the paper is organized as follows. Section 2 presents the background, including GPU architecture and unified virtual memory (UVM), TEE on the host side, and the AES-XTS encryption mode. Section 3 discusses the threat model of our work. Section 4 presents the design of the LITE and our proposed optimizations. Section 5 presents our experimental methodology, and Section 6 evaluates the results. Section 7 concludes our work.

2 Background and Related Work

2.1 GPU Architecture and Unified Memory

A GPU consists of an array of Streaming Multiprocessors (SMs), and each SM has its own control unit, register file, L1 cache, and software-managed shared memory [16, 22]. Multiple SMs share an on-chip L2 cache with multiple banks to provide high L2 access bandwidth. One or more L2 banks are then connected to a memory controller to provide high bandwidth to access device memory. With discrete GPUs, data is typically moved between CPU main memory and GPU device memory over the PCI-e interface.

GPUs use Single-Instruction Multiple-Thread (SIMT) architecture to achieve high throughput. As a result, GPUs can tolerate/hide long latency by leveraging massive thread-level parallelism. However, they tend to be sensitive to bandwidth due to the high number of concurrent threads.

Prior to UVM [16], programmers had to manage memory explicitly by allocating memory in the host and device memory and moving data between the host and the device. A way to automate this is to use direct store to move data from CPUs to GPUs by exploiting data producer-consumer relationship [28] . With UVM, programmers can avoid explicit memory management and rely on on-demand paging managed by UVM. UVM enables CPU and GPU to share the virtual memory space. Programs executed on the GPU are no longer limited by the size of device memory and can instead access host physical memory through a GPU virtual address.

2.2 Host-side TEE

A TEE may be designed to protect an application, a system, or memory. For the former, a ring-3 secure execution environment for code and data is provided by hardware (e.g., Intel SGX enclave) to protect against system software (OS and hypervisor) vulnerabilities as well as other code portions of the application that run outside the enclave. For the latter, a ring-0 secure execution environment is provided by hardware to protect a system (OS and applications) from vulnerabilities in the hypervisor or other systems. Examples include AMD Secure Encrypted Virtualization (SEV) [8] and Intel Multi-Key Total Memory Encryption (MKTME) [6]. AMD SEV and Intel MKTME are geared toward virtualized cloud computing servers where multiple virtual machines may share the same physical server. Each VM is provided a unique hardware ID with associated unique keys. Finally, the memory may be protected with encryption without regard to the software environment, such as in Intel Total Memory Encryption (TME) [6] and AMD Secure Memory Encryption (SME) [8]. TME is especially important for non-volatile memory with data remanence problems. A common support across all three types of execution environments is memory encryption. However, the type of memory encryption differs based on the goal of the memory security protection.

User-level enclaves such as SGX rely on counter-mode encryption and integrity verification relying on MACs and integrity tree [5]. Counter mode encryption is vulnerable to revealing plaintext of data if the counter can be changed or replayed by the adversary [25]. Thus, MACs and the integrity tree are an integral and necessary part of guaranteeing confidentiality in counter mode encryption, in addition to detecting integrity violations. In contrast, XTS mode encryption (shown in Figure 2) is often deployed without integrity verification as data confidentiality is not dependent on integrity verification. Thus, since the XTS encryption mode is not prone to revealing plaintext, integrity verification is only needed for detecting integrity violations. Without integrity verification, the attacker may modify the ciphertext of data in memory without being detected; however, the tampered ciphertext will be decrypted to an unpredictable plaintext value, which may be of little value to the attacker. Since Intel SGX usage is currently limited to a small portion of an application that is highly security sensitive, and our goal is to provide a TEE for a whole application or system, we instead focus on non-counter mode memory encryption.

Intel Total Memory Encryption (TME) [6] allows for the encryption of the entire physical memory of a system using the AES-XTS algorithm with a 128-bit key. The encryption key is generated using a hardware random number generator residing in the System-on-Chip (SoC). Multi-Key Total Memory Encryption (MKTME) [6] extends the TME to support multiple keys, and each page in the physical memory can be associated with a key. A process or the OS can read the plaintext of a page only when it has the right key in its page table entry for the page. The SoC supports a fixed number of encryption keys, and software can use the keys to encrypt any page in the memory.

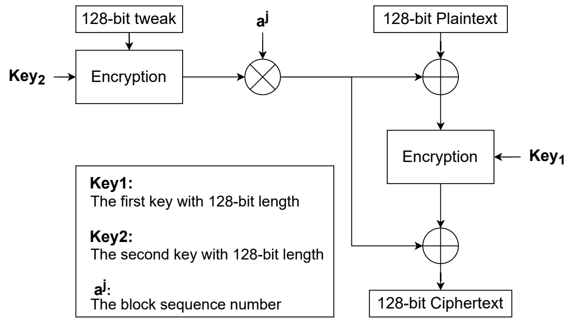

Figure 2. The block diagram of the AES-XTS encryption mode used in Intel MKTME.

Similarly, AMD Secure Memory Encryption (SME) [8] provides main memory encryption using a single key generated by the AMD Secure Processor (AMD-SP). Encryption is performed by AES encryption engines located in on-die memory controllers using a 128-bit key. AMD Secure Encrypted Virtualization (SEV) [8] extends SME by providing cryptographic isolation for a VM from the hypervisor and other VMs. This isolation is achieved by assigning a hardware tag and key to each VM and tagging pages for that VM. This assures that the plaintext of the VM pages to only be readable by the VM itself.

Figure 2 illustrates the AES-XTS with 128-bit keys and a tweak to encrypt a 128-bit plaintext. The tweak represents the address of the data being encrypted or decrypted [1] to ensure the same plaintext value at different addresses results in different ciphertexts. Compared to counter-mode encryption, the AES-XTS mode does not require counters to be kept to perform its operations, hence eliminating the counter integrity problem. Thus, other metadata (MACs and integrity tree) are also no longer needed to ensure confidentiality.

2.3 Related Work

There are a few recent proposals for TEE on GPUs. HIX [7] extended the Intel SGX interface to support the GPU enclave, which focuses on securing the GPU driver. The MMU design was also enhanced to prevent unauthorized access to the GPU memory-mapped I/O region. ZeroKernel [10] proposed a secure execution model that relies only on on-chip storage. This model assumes all of the kernel code can fit into the instruction cache and be stored there. It also assumes all of the PTE is cached in the TLB, then it removes all of the PTE from device memory and prevents page table reconstruction. These two proposals require no hardware changes to the GPU. Graviton [23] requires small hardware changes on the peripheral components. It assumes that the GPUs are using

3D stack memory, making it difficult to perform physical attacks. In Graviton, secure context isolation is achieved through an ownership tracking table. Using this table prevents unauthorized access to the victim address space. The focus of these three studies is to provide secure GPU execution without a lot of overheads, hence they did not include memory encryption.

Another approach to TEE is to bring CPU solutions to GPUs. The common counter scheme [15] uses counter-mode encryption and requires caches for the security metadata such as counter, MAC, Merkle Tree, and Common Counter Status Map. They observed that multiple counter values are updated simultaneously, resulting in the same value for several counters. They proposed to group several data blocks to have a common counter to reduce counter cache misses. Yuan et al. [26] analyzed the performance implication of counter mode encryption for secure GPU memory. They observed that the increase in memory traffic due to accessing security metadata, including counters and MACs, is the main contributor to performance degradation. The memory traffic increase would affect GPUs using non-volatile memory more than those using DRAM due to the lower bandwidth capacity for larger memory space and crash-recoverable ability [3]. In subsequent work, Yuan et al. [27] proposed the PSSM scheme to reduce bandwidth overhead from the metadata. None of the above works address the inter-operability of CPU/GPU TEEs, hence they will still suffer the high performance overheads when data crosses encryption domains.

Table 1 compares the solutions discussed above versus our software-based memory encryption LITE. Compared to the Common Counter and PSSM, LITE provides a unified encryption domain between CPU and GPU, allowing data to move from/to CPU TEE and GPU TEE without re-encryption. In contrast to Common Counter and PSSM, which require substantial hardware support (crypto engine per memory partition, metadata caches, etc.), LITE only requires small hardware support (write-once pages) for protecting kernel code. The flexibility of choosing an encryption algorithm is unique to LITE. Due to the flexibility of LITE, after establishing a common shared key, the GPU can use the same encryption scheme as the host CPU, leading to efficient communication between them. For example, unified virtual memory (UVM) can be supported, and data can be moved back and forth between CPU and GPU without re-encryption. LITE also has the flexibility of applying memory security to select data and select applications, depending on the need at the host CPU, and requires no re-encryption. Finally, since it requires little hardware support, LITE can be deployed easily in current production GPU. In contrast, Common Counter and PSSM solutions require much more hardware support and are difficult to deploy.

Table 1. Comparing LITE with prior GPU enclaves.

Aspect	Graviton [23]	Common Counter [15]	PSSM [27]	LITE
Memory encryption	No	Yes	Yes	**Yes**
Domain crossing	N/A	Yes	Yes	**No**
Hardware support	Low	High	High	**Low**
Flexible algo	N/A	No	No	**Yes**
Flexible app/data	N/A	No	No	**Yes**
Unified Memory	N/A	No	No	**Yes**
Deployability	Easy	Hard	Hard	**Easy**

3 Threat and Trust Model, Scope of Work

Both Intel TME and AMD SEV assume a threat model where the attacker has both a software and physical attack surface. The software attack surface is defined by the attacker having control over privileged software such as the OS and hypervisor. The physical attack surface is defined by the attacker having limited physical access to the machine to employ passive physical attacks such as snooping or scanning attacks but cannot modify data stored in memory. To be compatible with the host-side TEE, we assume the same attack model for GPUs.

We consider the following attacks to be out of the scope of this paper: active physical attacks (i.e., where data in memory is physically tampered with), side-channel attacks, and availability attacks.

Our trust model is as follows. We assume that the CPU, GPU, and CPU/GPU memories are trustworthy components in the sense that they operate correctly according to their specifications, free of design errors, faults, and trojan circuits. We assume that the system already has secure key storage in place, where the CPU and GPU have built-in public/private cryptographic key pair, with the public key readable from chip pins and the private key stored securely in a non-volatile manner in the CPU and GPU chips. Both chips are also assumed to have non-volatile storage to keep other keys, including session keys. Furthermore, we assume that a trusted party, such as the system integrator or the cloud administrator, has provided GPU public key to the CPU and CPU public key to the GPU. Alternatively, the CPU and GPU could automatically exchange public keys the first time they are connected. Thus, the CPU and GPU have a mechanism to initially trust each other.

Building on this trust, the CPU and GPU may initiate a Diffie-Hellman key exchange to establish a different shared session secret key, which enables a private communication channel between the CPU and GPU. If there is more than one CPU or GPU in the system, multiple secret keys must be tracked and stored on-chip. The shared secret key enables a private, authenticated communication channel that persists until the system shuts down. When the system is rebooted, the BIOS execution results in a new shared secret key to establish the communication channel.

Furthermore, our LITE scheme relies on a hardware feature where kernel code cannot be tampered with once it is loaded into GPU memory. Since active physical attacks are out of scope, the kernel code would not be altered by physical attacks. So, the possible threats would be malicious software, such as a driver. The code may be protected by the GPU page table, which sets the pages used for code as read-only.

4 The Design of LITE

In this section, we describe the design of LITE and discuss how, through a software solution with small hardware support, LITE allows GPU TEE to be co-designed with CPU TEE and provides great flexibility in which application or data is encrypted. Our discussion will start with rationale and overview, APIs, hardware support, and optimizations that enable LITE to achieve very low overheads for many applications.

4.1 Rationale and Overview

Figure 3 contrasts the typical data flow of current hardware-based GPU TEE design vs. our LITE. With current hardware GPU TEE (Figure 3(a)), because CPU and GPU TEEs use different encryption schemes and/or have their own encryption keys, they need to establish an intermediate ciphertext format that can be encrypted/decrypted by both the CPU and GPU. Data sent by the CPU must first be decrypted from the ciphertext C_1 in the CPU TEE domain and encrypted to the intermediate ciphertext form C_2 ①. After memory copy to device memory ②, the ciphertext needs to be re-encrypted again into the GPU TEE domain ③. When data is read by GPU, it is decrypted by the encryption engine ④ and stored in plaintext P in the on-chip GPU memory hierarchy. Notice the two sets of decryption and encryption that are added to the critical-path delay of CPU sending its data to GPU. This has to be repeated in the reverse direction as the GPU sends its computation result to the CPU at the end of kernel execution.

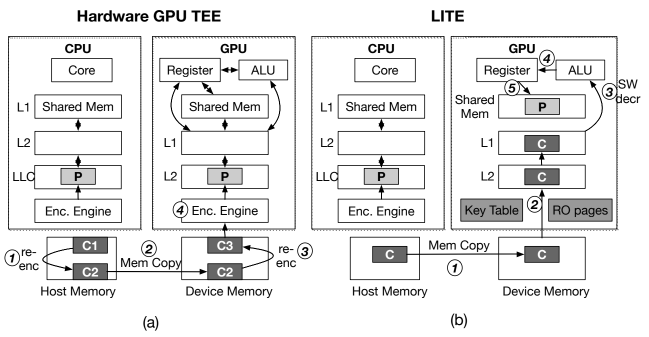

Figure 3. The LITE vs Hardware-based GPU TEE typical data flow from host to device.

In contrast, in LITE (Figure 3(b)), if the GPU is in the same encryption domain as the CPU, we eliminate the re-encryption of data between CPU and GPU. Data can be directly copied from host memory to device memory ①. In theory, one could co-design GPU memory encryption to match that of the CPU and also avoid cross-domain re-encryption. However, this is difficult in practice for several reasons. First, the GPU manufacturer may be different from the CPU manufacturer, making co-design difficult. Second, CPU may have different design and update cycles than GPU. For example, AMD Epyc 7251 processor uses XE-based encryption while AMD Epyc Embedded 3151 processor uses XEX-based encryption mode, although produced only eight months apart [24]. Finally, GPU may be used in many different scenarios to execute different workloads, hence ideally, GPU memory security should have a large degree of flexibility to match use cases and workloads. Therefore, in LITE, we focus on software memory encryption approach that can be deployed in GPU independently on CPU. A consequence of LITE's software approach is that data is brought into the GPU chip in ciphertext form ②. Then software decrypts data using ALU ③ operating on data on registers ④. Data may also be stored temporarily in plaintext in on-chip shared memory ⑤. LITE's software memory encryption approach is enabled in GPU because data movement into shared memory is controlled by software. The same approach is not deployable in CPU because data movement in caches is transparent to software.

There are several inherent advantages to LITE beyond avoiding domain-crossing re-encryption overheads. It can readily support unified virtual memory (UVM) because a page can be copied between host and device memory without any ciphertext transformation. Second, data is stored in ciphertext form in GPU caches, which are shared by multiple SMs that may run different kernels concurrently. While we are not aware of current security attacks that leak cached data, if such an attack arises in the future due to GPU design bugs, only the ciphertext leaks out.

Challenges in achieving LITE are numerous, and we seek to address them in this paper. Implementing the same encryption algorithm in software in GPU is only the first step in sharing an encryption domain with CPU TEE. First, simple APIs need to be defined for kernels to use. Second, another necessary ingredient is to share the same encryption metadata as used in the CPU TEE, including any tweak inputs (e.g., host physical address). Third, there needs to be hardware support at GPU to ensure that encryption software cannot be tampered with by the attacker and that encryption keys can be stored securely in GPU chip. Fourth, software encryption performance bottlenecks are aplenty and need to be addressed, including high latencies, warp divergence, and the challenges in collecting 128-bit of data from multiple threads to be encrypted/decrypted with block cipher algorithms. We will discuss them in the rest of the section.

4.2 Encryption APIs

Since AES is block-based encryption, data is encrypted by the host in blocks of 128 bits (if 128-bit AES is used). To decrypt data correctly, the same block must be gathered and decrypted. If the access pattern of the GPU is to contiguous data elements, then the encryption block can be gathered simply by collecting data from the registers of neighbouring threads. To do that, we rely on *shfl*, which is a warp-level shuffling instruction that enables a thread to read the registers of other threads. After decryption, data can be redistributed back to various threads by another round of shuffling. However, the if memory access pattern involves non-contiguous locations, assembling an encryption block is not as straightforward. In this case, we rely on padding so that each data item is expanded into a 128-bit single encryption block.

To ease adoption of LITE in the kernel code, we provide high-level AES APIs shown in Table 2. The first two calls are used for the case where adjacent threads in a warp access contiguous data, while the last two are provided when adjacent threads do not access contiguous data, hence padding is used (int_128 or a vector of four ints/floats) per thread.

Table 2. Encryption APIs

Interface to encryption and decryption
`encrypt(data, variable_addr, addr_type, enc_mode)`
`decrypt(variable_addr, addr_type, enc_mode)`
`encrypt_v4(data, variable_addr, addr_type, enc_mode)`
`decrypt_v4(variable_addr, addr_type, enc_mode)`

The decrypt/decrypt_v4 API loads the ciphertext and returns the plaintext. The encrypt/encrypt_v4 API takes the plaintext data in a register and returns the ciphertext generated using the key and the address tweak. In the last two arguments for each API function, the type of address (virtual or physical) and the encryption modes are specified to match those used by the CPU TEE. Listing 1 shows the implementation of APIs. `decrypt` shows the use of shuffling to assemble multiple contiguous 32-bit values in neighboring threads' registers into one 128-bit (Step 4), which is then decrypted (Step 5), and redistributed back to different threads' registers (Step 6). For `decrypt_v4` and `encrypt_v4`, with each thread accessing data using the float4 or int4 data types, data assembling/re-distribution is not needed, hence AES encryption function is invoked directly (Step 4 and 3, respectively).

Listing 1. Implementation of AES APIs.

```
unsigned int decrypt(unsigned int v_addr, bool addr_type, int enc_mode) {
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: accessing global memory at v_addr
    unsigned int temp = (unsigned int) (* v_addr);
    int_128 buff;
    unsigned int p_text //plaintext
    //step 4: assemble 128-bit data block
    if(tid % 4 == 0) { //tid is the thread id
      buff[0] = temp; //temp from thread i, i is a multiple of 4
      buff[1] = __shfl_down_sync(0xffffffff,temp, tid + 1) //temp from thread i+1
      buff[2] = __shfl_down_sync(0xffffffff,temp, tid + 2) //temp from thread i+2
      buff[3] = __shfl_down_sync(0xffffffff,temp, tid + 3) //temp from thread i+3
      //step 5: decrypt data with AES_decrypt
      buff = AES_decrypt(buff, v_addr);
    }
    //step 6: Distribute decrypted data to threads
    if(tid% 4 == 0)
      p_text = buff[0];
    else
      p_text = shfl_down_sync(0xffffffff, buff[tid%4], tid - tid%4]);
    return p_text;
}
unsigned int encrypt(unsigned int data, unsigned int v_addr, bool addr_type, int enc_mode) {
    unsigned int c_text; //ciphertext
    int_128 buff;
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: assemble 128-bit data block
    if(tid % 4 == 0) {
      buff[0] = data; //data from thread i, i is a multiple of 4
      buff[1] = __shfl_down_sync(0xffffffff,data, tid + 1) //data from thread i+1
      buff[2] = __shfl_down_sync(0xffffffff,data, tid + 2) //data from thread i+2
      buff[3] = __shfl_down_sync(0xffffffff,data, tid + 3) //data from thread i+3
    //step 4: encrypt data with AES_encrypt
    buff = AES_encrypt(buff, v_addr);
    }
    //step 5: distribute encrypted data to  threads
    if(tid % 4 == 0)
      c_text = buff[0];
    else
      c_text = shfl_down_sync(0xffffffff, buff[tid%4], tid - tid%4]);
    return c_text;
}
int_128 decrypt_v4(unsigned int v_addr, bool addr_type, int enc_mode) {
    int_128 p_text //plaintext
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: accessing global memory at v_addr
    int_128 temp = (int_128) (* v_addr);
    //step 4: decrypt data with AES_decrypt
      p_text = AES_decrypt(temp, v_addr);
    return p_text;
}
int_128 encrypt_v4(int_128 data, int_128 v_addr, bool addr_type, int enc_mode) {
    int_128 c_text //ciphertext;
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: encrypt data with AES_encrypt
    c_text = AES_encrypt(data, v_addr);
    return c_text;
}
```

4.3 AES Encryption and Address as the Tweak

As LITE encryption is based on software, the encryption algorithm and mode could be changed and updated to make it more secure, e.g., to add resistance to side-channel [13] or using a weaker algorithm such as DES [17] if higher performance is desired. For our implementation, we use the AES encryption from OpenSSL v1.1.1 and adapt it to CUDA [13]. It uses a T-table with a series of table lookups for each round and XORed to the round key.

To achieve CPU-GPU TEE interoperability, for host (CPU) memory encryption relying on a tweak, LITE needs to rely on the same tweak used by the host. Two approaches are possible. The first approach is to use the host physical address as the AES tweak. With this approach, the GPU needs to have the host physical address to decrypt data encrypted by the host and to encrypt the output for the host to decrypt. To achieve this, the GPU needs to keep the host physical address and uses it for decryption and encryption or be able to look it up. At least three techniques are possible. One technique is to add host physical address into the GPU page table such that given a virtual address, both host and GPU physical addresses can be looked up [19] [20]. With this technique, the GPU TLB also needs to be extended to include the host physical address. An alternative technique is for the GPU and host to share the same page table [11]. When the GPU misses in its TLB, the host IOMMU can be triggered to do a page table walk and provide the host physical address to the GPU for use in decryption and encryption. In this case, the GPU TLB can keep the host physical addresses but avoid modifications to the page table. Such a technique can utilize existing features such as NVIDIA Address Translation Service (ATS), available since Volta [21]. Finally, host physical addresses can be maintained in a separate data structure maintained entirely by the GPU. Such a structure needs to be populated prior to kernel launch, protected as read-only during kernel execution and looked up as needed by the GPU for encryption/decryption. In any case, in order for the host to decrypt the GPU computation result correctly, we need to reserve data pages in the host physical memory until the GPU computation is completed. In other words, LITE requires that all UVM data have consistent host physical addresses. It can be achieved by reserving & pinning host memory when UVM is allocated with the cudaMallocManaged API (a host function). If a page is first accessed by the GPU, the host physical address is sent through the page fault handler without data migration. If a page from UVM is migrated from host memory to GPU device memory, it is not unmapped, similar to page pinning. For the data generated from GPU, such as stack frames, for which there are no corresponding host physical addresses, the GPU physical addresses can be used as the tweak. This practice would not affect CPU-GPU interoperability since GPU private data is not part of UVM and would not be accessed by the host.

The second approach is to use virtual addresses as the AES tweak. As UVM provides a unified virtual address space between host and GPU, the virtual addresses are the same on either side. Therefore, no additional changes are needed to achieve interoperability. Neither pinning nor modifications to paging are necessary. However, most host memory encryption uses physical addresses, which requires some changes to host memory encryption design, e.g., additional datapath to pass virtual addresses from the core to the memory controller. In our experiments on real GPU hardware, we assume this approach for the purpose of performance evaluation. The performance of the first approach (i.e., using host physical address as a tweak) would be very similar to the second approach if the GPU TLB is expanded to include the host physical address.

4.4 Kernel Code Adaptation

With our provided APIs, GPU kernel modification is quite straightforward. After determining the encryption algorithm, its mode, tweak, and address used in the tweak, global memory accesses to the data are examined. By default, we treat all GPU global memory as secure (encrypted) unless configured otherwise. If the memory accesses for four consecutive threads fall into contiguous locations, then no padding is assumed, and the APIs to use are the first two in Table 2. Otherwise, padding is used, and the last two are used.

Kernel code can be adapted to use the APIs either manually or automatically by a compiler. With the compiler approach, the programmer can annotate the secure data variables through #pragma directives similar to OpenMP-style annotation, and the compiler transforms the directives into API calls. In this case, the compiler transformation forms a part of the trust base.

Listing 2 illustrates an example with tiled matrix multiplication. Global memory reads include `'a[row * n + (i*tile_size +tx)]'` and `'b[(i * tile_size * n + ty* n) + col]'`. Since accesses are to contiguous 32-bit data, we simply convert them to `'decrypt(&a[row * n + (i*tile_size+tx)], addr_type, enc_mode)'` and `'decrypt( &b[(i * tile_size * n + ty* n) + col], addr_type, enc_mode)'`. Similarly, the global memory store statement `'c[(row*n) + col] = temp_val'` is converted to `'c[(row*n) + col] = encrypt(temp_val, &c[(row*n) + col], addr_type, enc_mode)'`.

Listing 2. Matrix multiplication kernel after code adaptation.

```
__global__ void tiledMatrixMul(float *a, float *b,
    float *c, int n,
int tile_size) {
  __shared__ float A[SHMEM_SIZE];
  __shared__ float B[SHMEM_SIZE];
  int_128 buff1, buff2;
  int row = by * tile_size + ty;
  int col = bx * tile_size + tx;
  int temp_val = 0;
  bool addr_type = true; // true: phsyical, false:
    virtual
  enc_mode = 1; // 0:ECB, 1:AES-XTS, 2:XE, 3:XEX etc.
11    // Sweep tiles over entire matrix
12    for (int i = 0; i < (n / tile_size); i++) {
13      A[(ty * tile_size) + tx] = (float) decrypt( &a[row
       * n + (i * tile_size + tx)], addr_type, enc_mode);
14      B[(ty * tile_size) + tx] = (float) decrypt( &b[(i *
        tile_size * n + ty * n) + col], addr_type,
       enc_mode);
15      __syncthreads();
16      for (int j = 0; j < tile_size; j++) {
17        temp_val += A[(ty * tile_size) + j] * B[(j *
       tile_size) + tx];
18      }
19      __syncthreads();
20    }
21    c[(row * n) + col] = encrypt(temp_val, &c[(row * n) +
        col], addr_type, enc_mode);
22  }
```

Listing 3. BFS kernel before and after code adaptation.

```
1    struct pad_128 {
2       int32_t data;    //32 bit
3       int32_t pad;     //32 bit
4       int64_t pad;     //64 bit
5   };
6   bfs_kernel(pad_128 *row, pad_128 *col, pad_128 *d,
        float_128 *rho, pad_128 *cont,
7             const pad_128 num_nodes, const pad_128
        num_edges, const pad_128 dist)
8   {    ...
9   /*original code
10      for (int edge = start; edge < end; edge++) {
11          int w = col[edge];
12          if (d[w] < 0) {
13              (*cont) = 1;
14              ...
15          }  ... */
16  //code after padding
17      for (int edge = start; edge < end; edge++) {
18          int w = decrypt_v4(&col[edge], addr_type,
        enc_mode);
19          if (decrypt_v4(d[w], addr_type, enc_mode)<0){
20              int_128 i;
21              i.data = 1;
22              (*cont) = encrypt_v4(i, cont, addr_type,
        enc_mode);
23              ...
24          }
25  }
```

For non-contiguous access patterns, the data structure is padded before using the AES APIs to ensure that we would get 128-bit of data for encryption. The BFS function of the BC benchmark in Listing 3 shows `'col[edge]'`, `'d[w]'`, and `'(*cont)'` in lines 11-13 not accessing contiguous locations, hence the 'int' type is converted to pad_128 type, which contains 96-bit padding beside the 32-bit data. Padding, however, increases the data structure size and incurs significant performance penalties.

4.5 Hardware Support

In order for LITE to work securely, some hardware support is needed. The hardware support required by LITE includes on-chip key storage, remote attestation, and code-integrity protection. Among them, on-chip key storage and remote attestation are supported in the latest NVIDIA Hopper GPU [2]. In LITE, code-integrity protection is achieved by page table protection, which sets the code pages as read-only. Such hardware overhead does not affect the performance of kernel execution. Note that compared to Hopper GPU, which enables encrypted CPU-GPU data transfer, LITE provides confidentiality of data stored in GPU memory.

4.6 Code Optimizations

Masked shuffle. As discussed earlier, we could use *shfl* instruction to collect 128-bit of data for decryption. *shfl* has an implicit thread synchronization that acts as a barrier to ensure threads in a warp have finished earlier computation prior to data exchange. This fact is important when considering applications with branch divergence. If threads in a warp have thread-divergent branches and execute a *shfl*, which is permitted in Volta and later GPU architectures [12], the shuffle is delayed until the threads with the longest branch path complete. This performance bottleneck is illustrated in an example in Figure 4 (top), where odd-ID threads diverge on a branch path than even-ID threads. When only odd-ID threads need to load ciphertext from global memory, *shfl* forces them to wait for even-ID threads before data exchange in the decrypt function. To solve this, we propose a *masked shuffle* optimization that ensures only threads in the same branch path would be masked so that only those same threads will participate in data exchange. It takes advantage of the "mask" operand in the *shfl* instruction, which indicates specific threads in a warp that participate in the shuffling. We use the __ballot_sync() function to determine the set of participating threads in a warp for a branch path. This function is invoked before the diverging if statements. The result is used to set the mask of the shfl instruction such that only participating threads in a branch path perform the shuffle. In the example in Figure 4 (bottom), we use an odd mask in the decrypt function. By doing this, we let the odd ID threads complete the shuffle sooner, and the even threads do not participate in data exchange, thereby reducing the execution time.

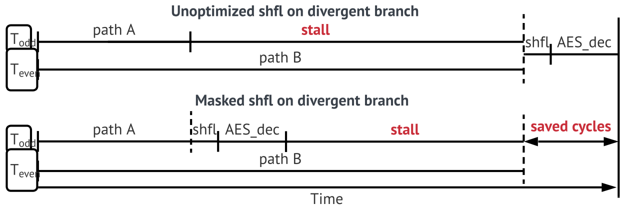

Figure 4. The timeline comparison between unoptimized shfl vs masked shfl. Following a divergent branch, only threads in path A fetch ciphertext from global memory.

Delayed Shuffle. Another problem with the implicit synchronization of *shfl* is that it limits instruction overlapping. For example, lines 13 and 14 in Listing 2 are two independent global memory reads, 'fetch A' and 'fetch B', and they can

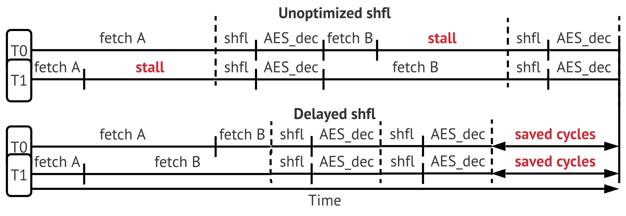

Figure 5. The timeline comparison between unoptimized code and optimized code with delayed shuffle. Fetch A and fetch B are two independent global memory accesses. Unoptimized code decrypts the data immediately while delayed shuffle decrypts the data after the data are loaded into shared memory/registers.

be issued back to back. However, due to the shfl used in the decrypt function, fetch A and fetch B are sequentialized, as illustrated in Figure 5 (top). As a result, when there is memory divergence, i.e., some threads having cache hits while others in the same warp have cache misses, the longest latency is exposed. To address this problem, we propose a *delayed shuffle* optimization, which delays the decrypt function after independent global memory reads. In the example of Listing 2, the decrypt function is moved to right before the syncthreads function in line 15. In other words, the ciphertexts are first loaded in shared memory (both fetch A and fetch B), then the decrypt function is used to overwrite the shared memory array with plaintext. This would re-enable overlapping between fetch A and fetch B, as illustrated in Figure 5 (bottom), leading to improved performance.

Selective Padding. If threads do not access contiguous data and the access pattern is hard to identify statically, there may be a data coherence problem that arises when two threads modify different data items in the same encryption block. We refer to this as an *encryption block false sharing* problem. Without hardware cache coherence, both threads will need to read the same encryption block in order to decrypt it, but this may create incoherent replicas. Earlier, we discussed that we eliminate the false sharing by padding the data structure such that each element is expanded to 128 bits. While this solution works, it increases memory footprint and bandwidth pressure. To improve on this, we note that having encryption block replicas only leads to coherence issues only if the block is modified. Thus, we perform *selective padding* optimization by differentiating the data access type; and skipping padding if data is read-only. The read-only attribute can be obtained from the programming model, e.g., the input buffers in OpenCL, compiler analysis, or programmer annotation.

5 Methodology

We evaluate LITE on an NVIDIA RTX 2080 GPU. The system runs on Ubuntu OS version 18.04 with NVIDIA driver version 440.33.01. For compilation, we use the CUDA version 11.0 and GCC version 7.1.0. We also use NVIDIA Nsight Compute to collect the hardware performance statistics. Each experiment was repeated 100 times, and we use the average of them.

To test LITE, we use a wide range of applications from two benchmark suites, Parboil and Pannotia, as shown in Table 3. They consist of both regular GPU and irregular GPU code. Parboil benchmarks have regular GPU code and work well with GPUs since their memory accesses can be coalesced, and a 128-bit block of data is accessed either by one or four adjacent threads. Pannotia has irregular GPU code that accesses memory locations depending on the input, i.e., the sequence of memory accesses are randomly determined by the input. Irregular GPU codes have relatively poor performance on GPUs. As discussed in the previous section, we use padding for the irregular benchmarks, in which a 128-bit data block is not accessed by a single thread or by four consecutive threads in a warp.

Table 3. Benchmarks

Name	Input	Suites	Reg	Size(MB)	Bottleneck
SPMV	1138_bus.mtx	Parboil[18]	Yes	0.04	Memory
MM	medium	Parboil	Yes	55.8	Compute
TPACF	small	Parboil	Yes	0.88	Compute
GRIDDING	small.uks	Parboil	Yes	63.7	Compute
MRI-Q	32_32_32_dataset	Parboil	Yes	0.44	Compute
LBM	short	Parboil	Yes	2.2	Memory
SAD	large	Parboil	Yes	8.2	Memory
STENCIL	128x128x32	Parboil	Yes	2.1	Memory
HISTO	large	Parboil	Yes	4.1	Memory
FW	256_16384.gr	Pannotia[4]	No	0.19	Memory
BC	1k_128k.gr	Pannotia	No	1.7	Memory
MIS	ecology1.gr	Pannotia	No	28.5	Memory
PAGERANK	coAuthorsDBLP.gr	Pannotia	No	12.6	Memory
SSSP	NY.gr	Pannotia	No	14.4	Memory
COLOR	ecology1.gr	Pannotia	No	28.5	Memory

In Table 3, the 'Reg' column indicates whether the benchmark is regular or irregular, 'Size' indicates the input size in megabytes, and 'Bottleneck' indicates the performance bottleneck of the benchmark is compute or memory. The bottleneck categorization is determined mainly from the DRAM utilization and Streaming Multiprocessor (SM) utilization, as shown in Figure 6. If the DRAM utilization is higher than the SM utilization, the benchmark is categorized as memory bound, and vice versa. An exception is SAD; this benchmark is categorized as memory bound since it has 98% utilization of the internal caches.

6 Evaluation

6.1 LITE Performance Overhead

We present the kernel execution time overheads incurred by naive vs. optimized LITE implementations over the unsecured GPU baseline in Figure 7 and Figure 8 for regular

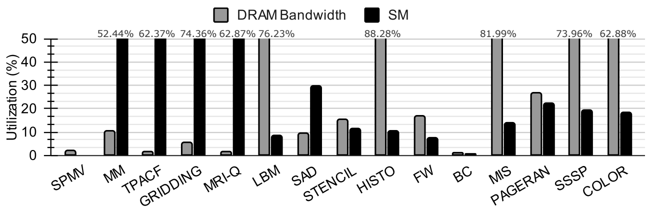

Figure 6. Streaming Multiprocessor (SM) utilization and DRAM bandwidth utilization.

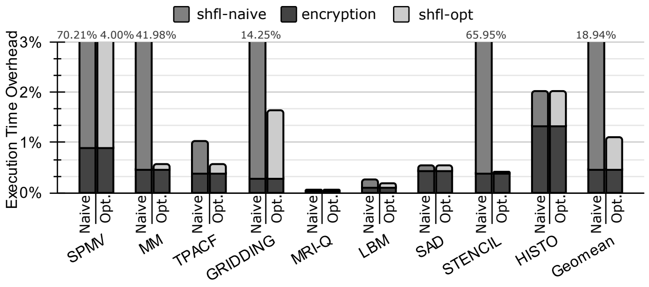

Figure 7. Kernel execution time overheads of naive LITE (shfl-naive+encryption) vs. optimized LITE (shfl-opt+encryption) over unsecured GPU baseline on regular GPU benchmarks.

and irregular code, respectively. The naive implementation always encrypts or decrypts data immediately after loading it or immediately before storing it to global memory. The optimized LITE applies all the optimizations described in Subsection 4.6. The execution time overhead is broken down into the time to perform encryption/decryption (*encryption*), the time to perform data shuffling without optimizations (*shfl-naive*), and the time to perform data shuffling with optimizations (*shfl-opt*). The last set of bars shows the geometric mean overheads across all benchmarks in the respective category. Note that the y-axes of the figure is capped at 3% for easier reading, but the actual magnitude overheads are shown in numbers for any bars that exceed the cap.

For regular applications in Figure 7, the naive implementation of LITE incurs 18.9% mean overheads.The overheads are especially very high for four benchmarks: SPMV (70.2%), MM (42.0%), GRIDDING (14.3%), and STENCIL (66.0%). In contrast, our optimizations are extremely effective, bringing the geometric mean down to more than one order of magnitude smaller, at 1.1%.

Examining the source of the overhead, we note that the encryption time itself causes only 0.5% overhead in general. Although encryption/decryption latency is relatively high when performed in software, GPU is good at hiding it via thread-level parallelism, resulting in nearly negligible encryption overheads. This observation is also consistent with the findings in the prior work [26]. However, anything that reduces the degree of thread-level parallelism can easily introduce high execution time overheads. In particular, the *shfl* instruction exposes the load imbalance of path divergence between threads in a wrap, forcing the wrap synchronization to wait for the slowest execution path to be completed. Also, it limits overlapping among independent instructions. The naive LITE shows that nearly all of the overheads for SPMV, MM, GRIDDING, and STENCIL, come from shuffling.

To verify if the implicit wrap synchronization in *shfl* is the culprit, we replaced the *shfl* instruction with *__syncwarp()* and noticed roughly the same overheads. We will now discuss the impact of optimizations on each benchmark.

For SPMV, we apply the masked shuffle optimization, ensuring that only threads that fetched data would participate in the corresponding data exchange and would be indicated active in the "mask". This optimization lowers the performance overhead from 70.2% to just 4.0%. The reason for the masked shuffle effectiveness is that SPMV may make some threads idle if their global thread id is beyond the length of the vector input. Thus, the idle threads should not be indicated as active in the "mask." We did not apply *delayed shuffle* because there are no independent data accesses.

For MM, we apply both the masked shuffle and delayed shuffle optimizations. We also replace some of the *shfl* with data fetch from the neighbouring addresses to collect the 128-bit encryption block. These optimizations significantly lower the overheads from 42.0% to just 0.6%, which is nearly two orders of magnitude improvement.

For GRIDDING and STENCIL, we applied both the masked shuffle and delayed shuffle optimizations. After applying the masked shuffle, the overhead decreases to 10.4% and 2.0% respectively, which is still somewhat high. However, after applying the delayed shuffle, the overheads go down to just 1.7% and 0.4% for GRIDDING and STENCIL, respectively. For TPACF, the masked shuffle optimization was applied similar to SPMV.

For irregular benchmarks (Figure 8), padding removes the false sharing coherence problem but significantly contributes to the overheads, incurring 206.5% geometric mean overheads for naive LITE due to increased working set size leading to higher bandwidth pressure. When we apply selective padding to the benchmarks, the execution time overheads decrease substantially as bandwidth pressure decreases. The geometric mean overhead decreases to 55.7%.

Only FW does not suffer from much overhead. The anomaly of FW is because its working set (Table 3) fits entirely in the L2 cache even after padding, hence padding does not increase bandwidth pressure. For other benchmarks, bandwidth pressure increases in padding due to the higher total number of memory accesses. In addition, miss rates often increase also. Figure 9 shows the L2 cache miss rate of unsecured GPU, full padding, and selective padding. Generally, the figure shows that padding increases the L2 miss rates,

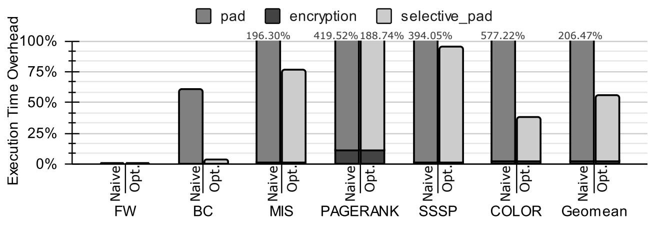

Figure 8. Kernel execution time overhead of naive LITE implementation (pad+encryption) vs. optimized LITE (selective_pad+encryption) over unsecured GPU baseline on irregular GPU workloads.

which are then reduced by selective padding. The only exception is COLOR. However, even though COLOR's L2 miss rate decreases with padding, its L1 cache miss rate increases (from 71% to 79%).

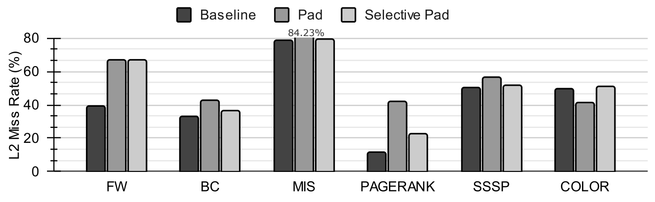

Figure 9. Comparison of the L2 miss rate of unsecure GPU (baseline) vs. full padding (pad) vs. selective padding (selective_pad) on irregular GPU code.

6.2 Benefit of Partial Encryption on LITE

Due to its software approach, LITE has great flexibility. We demonstrate one particular usage that stems from the flexibility, which is partial encryption. Here we explore scenarios where only the input or the output to GPU is confidential, hence needs encryption. Figure 10 shows the execution time overheads, normalized to full encryption (i.e., full encryption overheads are 100%), for input-only encryption with padding applied only to input, and output-only encryption with padding only applied to the output. Only irregular benchmarks are shown in the figure since full encryption overheads for regular workloads are already very small. The figure shows that partial encryption is generally effective, more so for input-only (82.0% lower) than for output-only (20.5% lower). Furthermore, its effect varies significantly across benchmarks, with some achieving nearly negligible overheads for input-only encryption (MS, PAGERANK, and COLOR) while others for output-only encryption (BC).

6.3 Performance of Software Implementation of AES

As LITE does not use a crypto engine to encrypt the data, it relies on software implementation of AES to perform encryption. The software performance of AES encryption of a single round of AES takes up to 1011 cycles before the cache warms up, while after that, it takes only 34 cycles for a single round. Similarly, the overall AES encryption latency is 7880 cycles before the cache warms up, while after that, it takes 340 cycles. The latency is measured by taking the difference between the clock cycle before and after each round or the encryption function. After the cache warms up, the GPU only needs to perform bit operations such as XOR, AND, and bit shift for encryption. While before the cache warms up, the GPU needs to fetch the T-table into its caches followed by bit operations for performing encryption, thereby incurring higher latency.

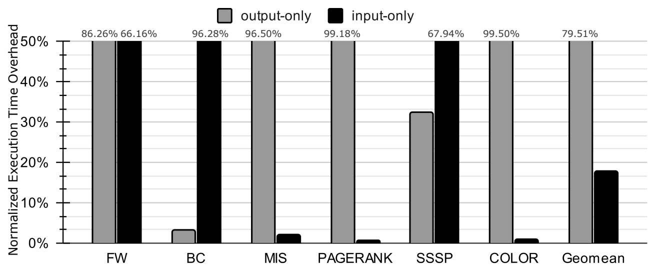

Figure 10. Normalized execution time overhead over full encryption of optimized LITE for input-only encryption vs output-only encryption on irregular GPU code.

6.4 Performance Comparison between LITE and PSSM

We modeled LITE in GPGPU-sim [9] and compared it with PSSM [27]. We also simulated the re-encryption process by running a re-encryption kernel for the input of each benchmark on the simulator. We simulated on all benchmarks listed in Table 3 up to 1B instructions or finished earlier. The results are shown in the Figure 11. As shown in the figure, for regular codes, LITE always achieved lower performance overhead compared to PSSM. For irregular codes, PSSM has lower performance overhead if the re-encryption overhead is not included. With re-encryption, PSSM shows a similar overall performance to LITE. Across all benchmarks, LITE achieved 11.7% performance overhead while PSSM with re-encryption achieved 68.8% overhead. From this, we could see that a key advantage of LITE is its interoperability, which eliminates the high re-encryption cost.

7 Conclusion

In this work, we proposed a software-based TEE for GPUs (LITE). We observed that when CPU and GPU TEE are not co-designed, communication between them incurs high performance overheads because of encryption domain crossing. Since LITE is software-based encryption, its encryption scheme can be co-designed to match host-side TEE, even after tape out. LITE only needs minor architecture support.

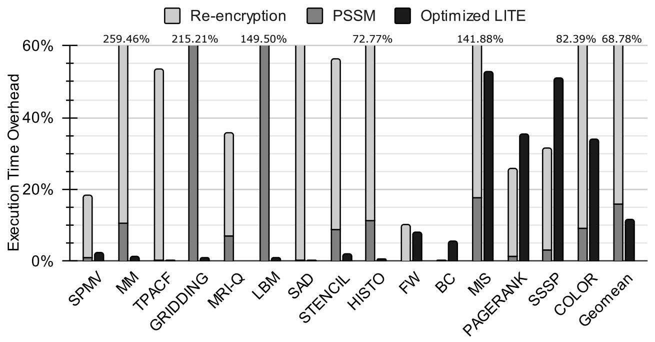

Figure 11. Comparison of execution time overhead of LITE vs. PSSM with re-encryption overhead evaluated in GPGPU-sim.

Measured on NVIDIA RTX 2080 GPU, naive LITE implementation incurs substantial performance overheads. We proposed three different optimizations including masked shuffle, delayed shuffle, and selective padding. Together, these optimizations are effective. LITE incurs execution time overheads of only 1.1% and 56% for regular and irregular benchmarks, respectively.

8 Acknowledgments

We thank the anonymous reviewers for their valuable comments. We also thank Rex McCrary for his feedback on this work. For this work, the UCF team is supported in part by NSF through grant 1908079 and by AMD, while the NCSU team is funded in part by NSF through grant 1908406 and by an AMD gift fund.

References

[1] 2019. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007)* (2019), 1–41. https://doi.org/10.1109/IEEESTD.2019.8637988

[2] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. 2022. *NVIDIA H100 Tensor Core GPU Architecture.* https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth

[3] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and Fast Lazy Persistency on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC).* 252–263. https://doi.org/10.1109/IISWC50251.2020.00032

[4] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC).* 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[5] Victor Costan and Srinivas Devadas. 2016. Intel sgx explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[6] Intel. 2019. Intel® Architecture Memory Encryption Technologies Specification. https://software.intel.com/content/dam/develop/external/us/en/documents/multi-key-total-memory-encryption-spec-753926.pdf

[7] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 455–468. https://doi.org/10.1145/3297858.3304021

[8] D. Kaplan, J. Powell, and T. Woller. 2016. AMD Memory Encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[9] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[10] O. Kwon, Y. Kim, J. Huh, and H. Yoon. 2019. ZeroKernel: Secure Context-isolated Execution on Commodity GPUs. *IEEE Transactions on Dependable and Secure Computing* (2019), 1–1. https://doi.org/10.1109/TDSC.2019.2946250

[11] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. 2021. Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1154–1168. https://doi.org/10.1145/3466752.3480083

[12] Yuan Lin and Vinod Grover. 2018. *Using CUDA Warp-Level Primitives.* https://developer.nvidia.com/blog/using-cuda-warp-level-primitives

[13] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. 2019. Scatter-and-gather revisited: High-performance side-channel-resistant AES on GPUs. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs.* 2–11.

[14] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016.* 1–9.

[15] Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, and Jaehyuk Huh. 2021. Common Counters: Compressed Encryption Counters for Secure GPU Memory. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* 1–13. https://doi.org/10.1109/HPCA51647.2021.00011

[16] NVIDIA Pascal. 2016. NVIDIA Tesla P100 Whitepaper.

[17] FIPS Pub. 1999. Data encryption standard (DES). *FIPS PUB* (1999), 46–3.

[18] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[19] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* USENIX Association, Philadelphia, PA, 109–120. https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki

[20] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2016. GPUvm: GPU Virtualization at the Hypervisor. *IEEE Trans. Comput.* 65, 9 (2016), 2752–2766. https://doi.org/10.1109/TC.2015.2506582

[21] Nvidia Team. 2022. *CUDA C++ Programming Guide.* https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[22] NVIDIA Tesla. 2017. NVIDIA TESLA V100 GPU Architecture Whitepaper.

[23] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[24] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1483–1496.

[25] Chenyu Yan, D. Englender, M. Prvulovic, B. Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 179–190. https://doi.org/10.1109/ISCA.2006.22

[26] Shougang Yuan, Ardhi Wiratama Baskara Yudha, Yan Solihin, and Huiyang Zhou. 2021. Analyzing Secure Memory Architecture for GPUs. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 59–69. https://doi.org/10.1109/ISPASS51385.2021.00017

[27] Shougang Yuan, Yan Solihin, and Huiyang Zhou. 2021. PSSM: Achieving Secure Memory for GPUs with Partitioned and Sectored Security Metadata. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS ’21)*. Association for Computing Machinery, New York, NY, USA, 139–151. https://doi.org/10.1145/3447818.3460374

[28] Ardhi Wiratama Baskara Yudha, Reza Pulungan, Henry Hoffmann, and Yan Solihin. 2020. A Simple Cache Coherence Scheme for Integrated CPU-GPU Systems. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218664