SAGE: Software-based Attestation for GPU Execution

Andrei Ivanov
ETH Zürich

Benjamin Rothenberger
ETH Zürich

Arnaud Dethise
KAUST

Marco Canini
KAUST

Torsten Hoefler
ETH Zürich

Adrian Perrig
ETH Zürich

Abstract

With the application of machine learning to security-critical and sensitive domains, there is a growing need for integrity and privacy in computation using accelerators, such as GPUs. Unfortunately, the support for trusted execution on GPUs is currently very limited – trusted execution on accelerators is particularly challenging since the attestation mechanism should not reduce performance.

Although hardware support for trusted execution on GPUs is emerging, we study purely software-based approaches for trusted GPU execution. A software-only approach offers distinct advantages: (1) complement hardware-based approaches, enhancing security especially when vulnerabilities in the hardware implementation degrade security, (2) operate on GPUs without hardware support for trusted execution, and (3) achieve security without reliance on secrets embedded in the hardware, which can be extracted as history has shown.

In this work, we present SAGE, a software-based attestation mechanism for GPU execution. SAGE enables secure code execution on NVIDIA GPUs of the Ampere architecture (A100), providing properties of code integrity and secrecy, computation integrity, as well as data integrity and secrecy – all in the presence of malicious code running on the GPU and CPU. Our evaluation demonstrates that SAGE is already practical today for executing code in a trustworthy way on GPUs without specific hardware support.

1 Introduction

Fueled by recent trends such as machine learning and the declining yields from Moore's Law, the use of accelerators to process the vast volumes of data is becoming indispensable. In fact, it is expected that the majority of compute cycles in public clouds will be executed on accelerators [29].

With the application of machine learning to security-critical or sensitive domains such as healthcare or financial modeling, there is a growing need for a mechanism that maintains integrity and secrecy for both code and data despite the computation being offloaded to the GPU.

With the wide-spread deployment of trusted execution environments (TEEs), e.g., Intel SGX [4] and ARM TrustZone [2], an important question is how security-sensitive computation tasks can be accomplished on GPUs. While first hardware-based TEEs on GPUs are starting to emerge [13, 20, 24, 27, 46, 49], how can we execute code securely on GPUs *in current environments*? As we have witnessed from the introduction of hardware-based TEEs on x86 platforms, it took over a decade until it became possible to fully and widely utilize these mechanisms. At the same time, technology progress in this space is a moving target as new attacks (among other factors) force vendors to phase out one specific hardware-based technology in favor of more robust successors (such as with the case of the deprecation of Intel SGX [39]). Given the importance of software executing on GPUs, it is clear that we need to find approaches to speed up the long lag time between deployment and wide-spread utilization.

A promising approach for bridging this gap is a software-only approach to trusted execution. In the context of CPU-based execution, a rich research field has contributed numerous approaches [8, 32, 33, 48]. The basic idea of the prior software-based or timing-based attestation approaches was to design a verification function that would run on an untrusted system and compute a checksum over itself – where both the correctness of the checksum and the time duration are measured by a trusted verifier. A correct checksum value that is returned before a threshold point in time, indicated to the verifier that the TEE was correctly set up and that the correct code is now executing (code integrity and launch point integrity). In combination with a system for control-flow verification, control-flow integrity can also be achieved.

The challenge of such software-based TEE establishment approaches lies in the creation of a verification function that will slow down noticeably or produce an incorrect checksum, if an adversary attempts to tamper with its execution.

The creation of a verification function for GPU environments poses numerous research challenges, which may be the reason why it has so far not been achieved, to the best of our knowledge. First and foremost, achieving (1) code se-

crecy and integrity, and (2) data secrecy and integrity, (3) in the presence of a malicious OS, (4) malicious code on GPU, and (5) a malicious CPU-GPU interconnect is a formidable challenge. Other challenges that we have to overcome include the absence of a true random number generator on the GPU, the lack of documentation from GPU vendors for a specific target architecture, no toolchain support to write native GPU microcode, and the difficulty in achieving optimal GPU utilization.

We design the SAGE system, which establishes a TEE on NVIDIA GPUs of the Ampere architecture (A100). SAGE utilizes an SGX enclave running on the host to act as a local verifier, and to bootstrap the software primitive to establish a dynamic root-of-trust (RoT) on the GPU. RoT establishment ensures either that the state of an untrusted system contains all and only content chosen by a trusted local verifier and the system code begins execution in that state, or that the verifier discovers the existence of unaccounted content. SAGE also sets up a shared secret key between the verifier and the GPU, which can be used to establish a secure channel to achieve integrity and secrecy for code and data transferred. Our results indicate that after a successful invocation of SAGE, the verifier obtains assurance that: (1) the user kernel on the untrusted device is unmodified; (2) the user kernel is invoked for execution on the untrusted GPU device; and (3) the user kernel is executed untampered, despite the potential presence of a malicious actor.

This paper presents the following contributions:

• We design a software-based attestation mechanism for GPU execution that enables secure code execution on NVIDIA Ampere GPUs, providing code integrity and secrecy, computation integrity, as well as data integrity and secrecy.

• We implement the race-condition TRNG and Verification Function used as basic security components in the software TEE. This requires an understanding of the GPU architecture and the format of the instructions used in the microcode, which we derive from our decoding and instruction generation framework.

• Through a proof-of-concept implementation on the NVIDIA A100 platform, we demonstrate the technical feasibility of the approach. Our implementation is publicly accessible at `https://github.com/spcl/sage`.

2 Background: GPU Fundamentals

In the following, we describe the fundamentals of NVIDIA GPUs and their programming model (CUDA) to illustrate how compute tasks are offloaded and executed on the GPU.

The GPU is connected via the PCI control engine to the host CPU and uses an internal bus for communication between its core components. The core components are the command processor, compute and DMA engines, and the memory system, consisting of a memory controller, registers, on-chip and device memory.

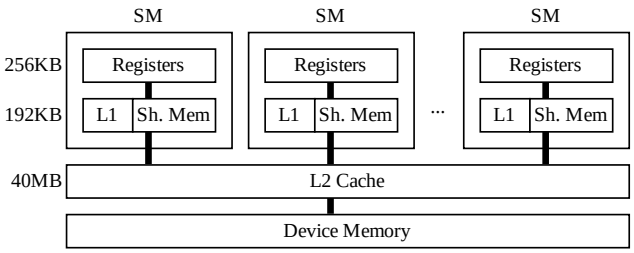

Figure 1: Memory hierarchy of a GPU with memory sizes of NVIDIA A100 GPU.

Controlling the GPU. Commands to the GPU are transmitted using a set of command queues known as *channels*. The GPU's command processor receives these commands and forwards them to the corresponding engines.

Data transfer to the GPU. GPU programming inevitably incurs data transfers between host and device memory. This is handled using direct memory access (DMA). The copy engine is responsible for handling DMA commands and their corresponding memory accesses.

GPU execution. The GPU's compute engine contains multiple Processor Clusters (PCs), each containing multiple Streaming Multiprocessors (SMs). SMs are partitioned into multiple processing blocks, each containing specialized processing cores (e.g., INT32 cores), a scheduler and a dispatch unit. *GPU kernels* to be executed on the GPU are scheduled to SMs and specify the number of threads to be created. These threads are organised in thread blocks and grids. Thread blocks are divided into warps. Each warp is a group of 32 parallel threads and gets scheduled by a warp scheduler.

Modern GPUs have multiple processing pipelines [23] for different data types. The FMA pipeline executes 32-bit floating point instructions and integer multiply and add (`IMAD`). The ALU pipeline executes 32-bit integer, logical, binary, and data movement operations. In addition, there are pipelines for 64-bit and 16-bit floating point, and Tensor core operations.

GPU memory system. The memory system on GPUs consists of a memory controller and different memory levels. The memory levels are associated to the compute system as follows (see Figure 1). Each processing block includes an L0 instruction cache and a register file. The combined processing blocks of a SM share a combined L1 data cache/shared memory that can be partitioned depending on the workload. Multiple SMs share an L2 cache before pulling data from global (off-chip) GDDR memory. *Registers* are a shared resource and are allocated among the thread blocks executing on a SM. Accessing a register consumes zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts. In case a thread requires more registers than available, the data contained in the registers is spilled into shared memory. *Shared memory* is not only used for register spilling, but also enables communication and memory reuse between threads in a block.

3 Problem Definition

We first describe the design goals we strive to achieve, as well as the assumptions and the adversary model we consider.

3.1 Design Goals

Verifiable code execution on the GPU. Verifiable code execution describes the problem in which a verifier wants a guarantee that some arbitrary code has executed untampered on an untrusted platform, despite the potential presence of a malicious entity (e.g., malicious software) [32]. This problem is typically approached by verifying code integrity through root of trust attestation, setting up an untampered code execution environment, and then executing the code.

Data integrity and confidentiality. In addition to code integrity also the integrity and/or confidentiality of the data executed on the GPU must be ensured. Specifically, we aim to guarantee that the adversary cannot observe or tamper with data transferred to/from the GPU by a trusted application that runs in a CPU TEE.

Dynamic root of trust without hardware support. Dynamic root of trust establishment denotes the problem of dynamically setting up a trusted computing base (TCB) on an untrusted platform without hardware support. All code contained in the dynamic root of trust is guaranteed to be unmodified and it can thus be used to provide externally verifiable code execution.

3.2 Assumptions

Verifier and GPU on the same machine. We assume that the verifier is executed on the same machine as the GPU we want to attest. The GPU is directly connected to the host CPU over a bus (e.g., PCIe with a latency of ~500 ns [18]).

GPU hardware configuration. We assume that the verifier knows the exact hardware configuration of the GPU, including the GPU model, the number of cores, the memory architecture, and the GPU clock speed. This assumption is practical when the hardware configuration is managed by the user or a trusted cloud provider. The machine owner has knowledge of the hardware configuration, which cannot be altered by software. In this configuration, we aim to protect against remote attackers who may arbitrarily modify software.

3.3 Threat Model

In the following, we discuss the threat model by defining the trusted compute base (TCB) and outlining the capabilities of an adversary. The TCB of a system refers to all hardware and software components that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system.

Trusted compute base (TCB). We assume that the remote adversary has full control over the software of the untrusted host system. In other words, the adversary has administrative privileges, can tamper with the operating system, or the guest operating system and the hypervisor in case of virtualization. However, we assume that the hardware primitives of the CPU and GPU, including firmware are contained in the TCB. Since SAGE uses Intel SGX, it inherits the TCB of SGX (which includes the CPU package, trusted libraries, etc.).

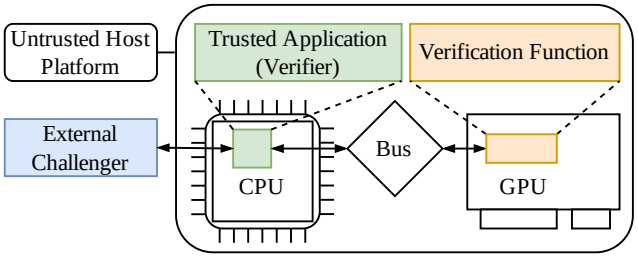

Figure 2: Abstract system model.

Capabilities. Considering these capabilities, an adversary can read and tamper with code or data of any victim process, and can access or modify data in DMA buffers or commands submitted to the GPU. Furthermore, the adversary could inject packets in arbitrary locations on the I/O communication path between the host and the GPU. This gives the adversary control over attributes, such as the address of GPU kernels being executed and parameters passed to the kernels. The adversary may also access device memory directly over MMIO, or map a user's GPU context memory space to a channel controlled by the adversary. In GPUs that support multi-tasking, malicious kernels can be dispatched to the GPU, thereby accessing memory belonging to a victim's GPU context.

Out of scope. Since this work tackles the problem of trusted execution on the GPU, we do not consider attacks that target SGX, such as physical attacks to the CPU package or side-channel attacks on SGX. In addition, we do not consider system availability attacks that prevent the execution of our process, as an adversary with the described capabilities can always prevent the deployment of computing tasks on the GPU. We assume that the adversary is not capable of using undocumented GPU capabilities to execute an attack. We believe that it is the responsibility of the manufacturer to ensure that such attacks are not possible, since the details of hardware and driver implementations are hidden from users.

4 SAGE Overview

SAGE addresses the problem of verifiable code execution on a GPU without hardware support, in which the verifier wants a guarantee that the user kernel has executed untampered on an untrusted GPU platform, even in the presence of an adversary. Figure 2 illustrates the abstract system model we consider.

SAGE comprises two main components. The first component is the verifier, which runs as a trusted application on the host CPU (e.g., using Intel SGX [4]) and is attested by an external challenger. The second component is the verification function (VF), which runs on the the untrusted GPU. The VF computes a checksum over its own code, and is constructed in an intricate way such that if a change is applied to the VF then either the execution will slow down in an externally

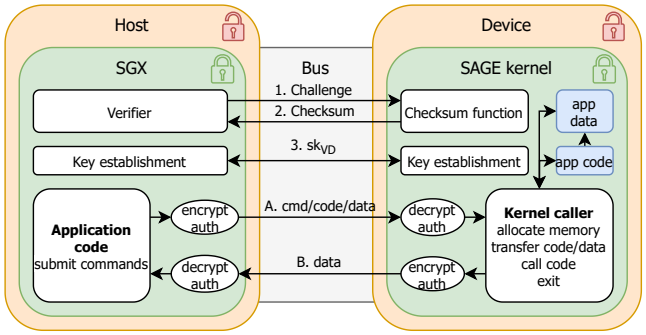

Figure 3: Overview of SAGE. The numbers represent temporal ordering of events. The letters show repetitive operations.

detectable manner, or the checksum value will be incorrect.

The verifier dispatches to the GPU the VF and then invokes it with a challenge while measuring the VF execution time. The VF computes a checksum value and returns it to the verifier. Using the same VF logic, the verifier independently computes and verifies the correctness of the checksum value. If the checksum returned by the VF is correct *and* it is returned within the expected time, the verifier obtains a guarantee that a dynamic root of trust on the GPU was established.

Once the dynamic root of trust has been established, the VF sets up an untampered execution environment. During the setup of the execution environment, a shared key between the verifier enclave and GPU is established; afterwards, only commands authenticated with this key are accepted, including the movement of encrypted kernel code and data between host and GPU. SAGE guarantees execution integrity and memory protection for the code and data stored in the GPU memory (see §8). Figure 3 shows an overview of SAGE including a sequence of events.

5 Verification Function (VF)

The VF that runs on the untrusted GPU is the fundamental component of SAGE. We now describe in detail these tasks and the challenges they entail.

5.1 Design Requirements

The VF must be carefully constructed in such a way that if an adversary were to tamper with the VF or the user kernel, it would result in either a wrong checksum or a noticeable time delay. Before offering a concrete design for the VF, we describe several required properties and outline how these properties influence the correctness of the checksum or the VF execution time. We defer our security analysis to §8; the following properties also account for the attack surface analyzed therein.

Time-optimal implementation. The implementation of the VF must be *time-optimal*. Otherwise, the adversary could use a faster implementation and use the time saved to forge the checksum (e.g., by injecting instructions).

Maximize resource usage during checksum computation. To prevent the adversary from running any other computation during the checksum computation, the VF maximizes its resource usage on the GPU by using all available SMs and avoiding "empty" threads. Moreover, each thread should use the maximum number of available registers to prevent the adversary from using those registers. Thus, if a malicious computation attempts to use more registers than available, the values of the affected registers are spilled into shared memory, resulting in a noticeable execution time difference (4- vs. 30-cycle latency for registers and shared memory, resp.).

Predictable execution time. The execution on GPUs is optimized to achieve high data throughput with deterministic latency, but the execution time is non-deterministic (e.g., due to multi-threaded execution, scheduling, and caching). The VF execution time should have low variance so that the verifier can predictably determine the execution time.

Challenge-dependent checksums. To prevent the adversary from pre-computing the checksum before making changes to the VF, and to prevent the replay of old checksum values, the checksum needs to depend on an unpredictable challenge sent by the verifier.

5.2 Concrete VF Design

The VF consists of initialization, self-verifying checksum function to establish a dynamic root-of-trust, and establishing an untampered execution environment including a key establishment protocol between the verifier and the GPU.

During the initialization phase, the memory buffer is allocated on the GPU and returned to the verifier. Then the VF code is copied into the buffer.

5.2.1 Self-Verifying Checksum Function

The checksum function is used to obtain a guarantee that the integrity of the VF code running on the GPU is unaffected by an adversary. For this purpose, the checksum function computes a checksum over the entire VF code. The resulting checksum can be used as a *fingerprint* of the VF and enables detection of changes to the VF code. If an adversary modifies the VF code, the checksum will differ with high probability. Thus, once the verifier receives a correct checksum within a threshold time, it has a guarantee that the VF code running on the GPU is unmodified.

Since the checksum computation code is part of the VF and will thus be included in the checksum calculation, the checksum function computes the checksum over its own instruction sequence and verifies itself. This property is further referred to as *self-verification*.

Checksum initialization. GPUs contain multiple multiprocessors that can be used for parallel execution. To achieve the maximal computational power of a GPU, the verifier sends a set of challenges containing a specific challenge value for each multiprocessor. Upon receiving a set of challenges, each multiprocessor uses its challenge as a seed value to initialize all per-thread state with pseudo-random data. Each thread has its own set of registers which are used to store the run-

ning checksum values and a data pointer. The data pointer references the VF code in the initially allocated buffer.

Checksum loop. The checksum computation is performed iteratively. Each iteration executes the same number and type of instructions and has a constant execution time.

Pseudo-random memory access prevents the adversary from predicting which instruction will read the potentially-modified memory location and forces the adversary to monitor every memory read by the checksum code, resulting in a noticeable time overhead. Indirectly, this process performs the inclusion of the data pointer in the checksum to prevent memory copy attacks (see §8).

Update the checksum. The running checksum values are updated to include the accessed VF code into the checksum value using a sequence of instructions. To achieve a time-optimal implementation, we use simple arithmetic and logical instructions (e.g., +, <<, >>, etc.) that are challenging to implement faster or with fewer operations. Taking inspiration from the strong ordering in [32], the instructions used to update the checksum alternate between arithmetic and logical instructions to enforce a strong ordering of the instructions.

Self-modifying code. The instructions of the self-modifying code fragment depend on current value of the checksum and are changed in each iteration of the checksum function. In our case the current value of the checksum function is used as an immediate value for an instruction (see §6.5 for details).

Checksum epilogue. Since the checksum computation is conducted using individual threads located on different multi-processors, the checksum values need to be aggregated before sending the checksum result back to the verifier. This aggregation is conducted in three steps. First, we aggregate the checksum per warp. Each of the per-thread checksums is added pairwise to obtain a warp-level checksum. Second, the warp-level checksums are aggregated by thread block using shared memory. Finally, we aggregate the checksum per grid using global memory. Each of the aggregation steps uses a pairwise addition (which is mapped to an atomic add instruction in native assembly). The final result of the checksum computation is then sent to the verifier.

5.2.2 Untampered Execution Environment

After establishing a dynamic root-of-trust on the device, the VF sets up an execution environment in which the user kernel is guaranteed to run untampered. This includes setting up a shared secret between the verifier and the device, and checking the authenticity of the user kernel to be executed on the GPU using a hash function. The shared secret can then be used to authenticate and encrypt commands and data sent by the verifier to the device and vice versa.

Key establishment. To establish a shared secret between the verifier and the device, we rely on the SAKE protocol [31], a protocol for key establishment between neighboring nodes in sensor networks without requiring any prior secrets. The protocol is based on the Diffie-Hellman key exchange protocol and uses the Guy Fawkes protocol [1] for authentication. The Guy Fawkes protocol is based on hash chains and relies on the property that each of the participants needs to authenticate the other party's hash chain. In SAKE, this authentication is achieved using software-based attestation and exploits the asymmetry in the computing time between the genuine checksum function executing on the device and an external entity computing the checksum value. This allows us to use the resulting checksum as a short-lived secret. Furthermore, the SAKE protocol assumes that the adversary does not introduce any computationally more powerful nodes into the network, which aligns with the assumptions for SAGE (see §3.2).

To apply the SAKE protocol to SAGE, we change the protocol as follows: 1) The checksum function in SAKE that was proposed for the use in sensor networks is replaced with SAGE's checksum function. 2) Instead of both participants acting as challengers, only the host enclave will engage as a challenger. 3) We replace the cryptographic primitives used in the protocol with AES-CMAC as the MAC function and SHA256 as the hash function.

The key establishment protocol in SAGE works as follows. First, the verifier sets up its own hash chain for the Guy Fawkes protocols and DH public key as:

$$V: \quad v_0 = g^a \bmod p \quad v_1 = H(v_0) \quad v_2 = H(v_1) \tag{1}$$

where a is a random bitstring $a \leftarrow^{\mathcal{R}} \{0,1\}^n$ Then, it sends v_2 to the device and records the current time as t_0.

$$[t_0] \quad V \rightarrow D : v_2 \tag{2}$$

Upon receiving v_2, the device uses it as a challenge for the checksum function and then uses the computed checksum and a random value to generate its own hash chain and replies to the verifier:

$$D: \quad w_0 = H(c \,\|\, r) \quad w_1 = H(w_0) \quad w_2 = H(w_1) \tag{3}$$

where r is a random bitstring $r \leftarrow^{\mathcal{R}} \{0,1\}^n$, c is the result of the checksum computation and $\|$ denotes to concatenation.

$$[t_1] \quad D \rightarrow V : w_2, \ \mathrm{MAC}_c(w_2) \tag{4}$$

The verifier checks if the measured execution time $(t_1 - t_0)$ matches the expected execution time and aborts the protocol otherwise. In the meantime, the device sets up its own DH public key:

$$D: \quad b \leftarrow^{\mathcal{R}} \{0,1\}^n \quad k = g^b \bmod p \tag{5}$$

Then, the verifier and the device gradually disclose the remaining of their hash chains to each other:

$$V \rightarrow D : v_1 \qquad D \rightarrow V : w_1, \ k, \ \mathrm{MAC}_{w_0}(k) \tag{6}$$

$$V \rightarrow D : v_0 \qquad D \rightarrow V : r \tag{7}$$

For each message the recipient checks whether the received value matches the expected hash chain. Finally, the verifier V and the device D compute the shared secret key sk_{VD}:

$$sk_{VD} = k^a = (g^b)^a \bmod p \quad sk_{VD} = v_0^b = (g^a)^b \bmod p \tag{8}$$

After the dynamic RoT has been established on the GPU and the integrity of the user kernel has been checked, the host enclave can start transferring code and data to the GPU. Depending on the sensitivity and security criticality of the domains, the data could be either *authenticated* and/or *encrypted* using the established symmetric key sk_{VD}.

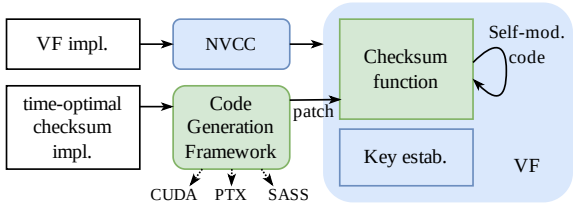

Figure 4: Code pipeline to generate the VF microcode. The green blocks generated using our framework.

6 Implementation

The requirements to achieve a time-optimal (see §5.1) implementation on the Ampere architecture (further discussed in §6.3) include maximizing GPU utilization, consuming all available compute resources, optimally filling the processing pipelines, and optimizing cache usage.

Unlike the higher levels of the CUDA computing platform such as the CUDA C++ language extension and the parallel thread execution (PTX) virtual machine and instruction set architecture, NVIDIA provides very little information about the hardware-specific instruction sets for a specific target architecture. Moreover, even if one resorts to write inline PTX virtual assembly, the Streaming (or Shader) Assembler (SASS) code emitted by the compiler often does not achieve the performance of native GPU applications. The execution of microcode that has been compiled using the regular CUDA compiler often is on the order of 10x slower compared to optimized microcode [14, 15]. As a consequence, libraries used for high-performance computing (e.g., cuBLAS [25]) contain highly optimized microcode tailored to a specific architecture. In addition to the performance gap to native GPU code, the user has no control over the translation from PTX virtual assembly to the SASS assembly for the target architecture.

To achieve a time-optimal implementation, we needed to implement a custom instruction generation framework that allows patching of binary microcode with a highly optimized version. The implementation of this framework requires understanding the Ampere architecture and the instruction format used in microcode. Although our focus in this paper is on the A100, we expect that small modifications to the code generator can provide support for the Volta and Turing architectures as well. Figure 4 illustrates the pipeline used to generate the VF. The VF is implemented using CUDA C++ and compiled using `NVCC`. However, the section containing the checksum function is patched using an optimized implementation generated as binary microcode using our framework.

6.1 Instruction Decoding

To understand the instruction format used in the recent Ampere GPU architectures, we implemented a framework that allows decoding of instructions using `cuobjdump` and `nvdisasm` [21] by decoding handcrafted code samples and samples from existing CUDA libraries (e.g., cuBLAS [25]).

Instruction format. NVIDIA's Ampere architecture adopts the same general instruction format as its predecessors Turing and Volta [14, 15]. All these architectures use 128 bits to encode both an instruction and its associated scheduling control information. The encoding that is used in these architectures is fixed length and uses similar encodings for all instructions. Figure 5 illustrates a typical instruction encoding.

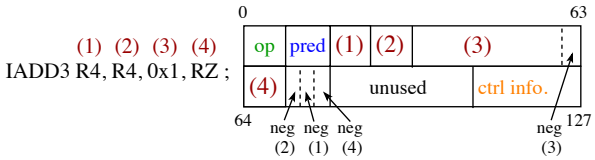

Figure 5: Instruction as decoded by `nvdisasm` and its format. `pred` denotes predicates, `op` refers to the operation code, and `neg` allows negating the corresponding parameter.

Control information. The control information section in the instruction encodes scheduling decisions taken by the compiler that the hardware must enforce. The control information is organized as follows: reuse flags (4 b), wait barrier mask (6 b), read barrier index (3 b), write barrier index (3 b), yield flag (1 b), and the number of stall cycles (4 b). The reuse flags allow data reuse between instructions without accessing any register ports. The wait barrier mask and indices are used for instructions with variable latency (e.g., instructions involving a memory access). These dependency barriers can be used to enforce the completion of variable-latency instructions. The yield flag is used to balance the workload assigned to a processing block. The stall cycles indicate the latency of the instruction before issuing the next instruction. Jia et al. present a detailed description of the control information [14].

6.2 Instruction Generation

Understanding the instruction format allows us to generate the specific instructions we need for our implementation. These instructions then need to be translated to the correct binary format. For this purpose, we implement an instruction generation framework that allows emitting instructions either in CUDA C++, the virtual assembly language PTX, or as binary microcode that is natively executed on the GPU.

The instruction can be defined in the following format, where the section separated using `|` symbol describes the control information for the instruction (barrier mask B, read barrier index R, write barrier index W, yield flag Y, and number of stall cycles S):

```
B......|R.|W.|Y1|S1| IMAD.U32 R28, R28, 2048, R28;
```

Our instruction generation framework then translates the instruction to the selected target language (CUDA C++, PTX, microcode). This allows us to rapidly prototype checksum functions and compare performance between implementations in each of the languages.

6.3 Time-optimal Technical Requirements

We formulate the following technical requirements for a time-optimal implementation of the checksum function. These are subject to characteristics of the target architecture; in our case, the NVIDIA Ampere architecture.

Maximize resource consumption. To maximize the resource consumption during the checksum computation, the checksum function must use all available compute resources. The NVIDIA A100 GPU has 108 Streaming Multiprocessors (SMs) each containing 64 `FP32` and 64 `INT32` units [19] that must be used during each clock cycle.

Optimally fill FMA and ALU pipelines. Since both the FMA and ALU pipelines have an instruction issuing latency of 2 clock cycles, `FP32` and `INT32` instructions must be interleaved to fully saturate both pipelines. In addition, instructions that use registers with a direct dependency must be executed with a latency of at least 4 clock cycles to avoid pipeline stalls (e.g., read-after-write dependency).

Optimal GPU utilization. To achieve full GPU utilization, the number of threads per thread block needs to be picked according to the target architecture. The A100 achieves full GPU occupancy by assigning 2 blocks of size 1024 to all the 108 available SMs (216 total). Each SM has 65,536 32-bit registers available for threads. To use all registers during the checksum computation while maintaining full utilization of the GPU, 32 registers are assigned per thread [22].

Cache size. The code blocks should not exceed the capacity of L0 and L1 instruction caches (see Figure 1).

6.4 Selection of Optimal Overheads

An optimal implementation of a checksum function should perform a useful computation step in each clock cycle. In practice, this requires a highly optimized use of the underlying hardware. In the following, we show a recipe for building such a checksum function for the A100 GPU.

Unutilized clock cycles are mainly caused by instruction cache misses, global memory access latency, pipeline stalls, and jumps. In the beginning of each clock cycle, the SM warp scheduler selects a subset of warps (up to `S=4` on A100) from all active warps (up to `A=64` on A100) to execute. This selection mechanism can avoid performance losses if at least `S` are ready to execute on each clock cycle.

To analyze the performance of the checksum function, we use a simplified model of the number of clock cycles per instruction. In the following, we will demonstrate that it helps to reach the performance with the precisely specified number of clock cycles. We distinguish the total number of useful clock cycles `X` and overhead cycles `Y`, so that the total number of clock cycles spent by the code using a single thread is `X+Y`. For example, with proper instruction ordering to avoid pipeline stalls, an `IMAD` instruction has `X=1` and `Y=0`. An instruction reading from global memory has `X=1` and approximately `Y=250`. To prevent attacks on the checksum function by executing some instruction each clock, the value of `Y` must not exceed `X(A/S-1)`. Then, the GPU scheduler will be able to completely hide the overhead `Y` so that the actual amount of time spent will be `X`.

Integer shifts and multiplications with addition directly affect the result of the checksum calculation. However, the instruction to jump from the end of the loop body to its beginning does not change the checksum. The attacker may try to unroll a few iterations of the loop to save the clock cycles required to perform this jump (and potentially misuse them for an attack). To prevent such attacks, we unroll the loops until it is not possible to unroll them further without causing instruction cache misses. The target value `Y` for unrolling must be so large that one additional instruction cache miss will increase it to `Y′` without the possibility for a hardware scheduler to compensate for the increase (and potentially hide it) using scheduling.

In practice, we have noticed that achieving this level of control over the order of instructions, and the arrangement of unrolled loops is very difficult without vendor support: the documentation on SASS and hardware details is deliberately kept closed to reduce backward-compatibility issues. It is especially difficult to control instruction cache misses because of the use of self-modifying code to protect against memory copy attacks. The only way to invalidate the instruction cache on the A100 is to overflow it with the block of instructions of the cache size, so controlling the value of `Y` by changing the size of the checksum function is not possible. That leaves only memory accesses and jumps that can change `Y`. We assume that adding an instruction to invalidate the instruction cache requires minimal (or no) changes to the GPU architecture because a similar instruction already exists for the data cache (`discard` in PTX ISA or `CCTL` in SASS).

6.5 Implementation of SAGE

Verifier. We implement the verifier enclave using the Intel SGX SDK [11] and its `tcrypto` library [10]. The enclave creates a CUDA context on the GPU, loads the VF as a module, and calls the VF kernel. To generate nonces in the enclave that are then transferred to the GPU as challenges, we use AES-CTR with an IV that has been generated using a TRNG during the enclave creation.

VF. The VF is implemented in CUDA C++, except the checksum function component, which is patched by binary microcode using our framework. The checksum function executes a loop containing the following operations.

First, the iteration counter is increased and checked if the maximum number of iterations is reached. Then the VF data block `D` is read from memory from the location defined by the current checksum value `C`, used as an offset: `D=data_ptr+(4×C mod data_size)`. After the load is complete, it is included in the checksum `C+=D`.

The read from main memory may take 250 – 500 cycles to be completed. The GPU compiler sets a read barrier for this instruction and the GPU stalls the compute pipeline until

the read has been completed. Instead of the stall, we develop an instruction pattern that is executed while waiting for the memory read to complete ("busy waiting"). We use interleaved (see §6.3) `X+=X<<N` with `IMAD` (FMA) and `X+=X>>N` with `LEA.HI` (ALU) instructions where `X` is any of 32 registers. The security of this computation depends on the existence of an alternative sequence of instructions which can compute the result faster. We expect that for some cases of long sequences or poorly chosen shift amounts, it is possible to find a shortcut constructed similarly to the jump ahead function in the xorshift pseudo-random number generator (PRNG) [45]. To prevent such shortcuts, we partition long sequences by requiring materialization of intermediate values, breaking them with random memory accesses included in the checksum. We aim for sequences of such length that the cost of implementing a shortcut is higher than performing the actual computation.

After updating the checksum function, we compute the self-modifying code that consists of the following binary instruction: `C+=C>>N`, where the immediate `N` depends on the current checksum value. We overwrite immediate parameter with the current value of the checksum. Thus, the value of `N` changes for each iteration and ensures that we are executing the code that we are verifying. To avoid race conditions when updating the immediate value of these instructions, these instructions are required to be located in different memory areas for each thread block.

6.6 Random Number Generation on GPUs

For the key establishment protocol based on the modified SAKE protocol, the GPU needs to be able to generate random values. Given that the adversary knows the entire code executing on the GPU, we cannot use a secret provided by verifier to initialize the PRNG used in the protocol, but instead must rely on a true random number generator (TRNG).

TRNG implementation on GPUs. Approaches that use physical unclonable functions (PUFs) to initialize PRNGs on the GPU [7, 30, 43] are not practical to be used in SAGE as they either require resetting the GPU or use features that are under control of the adversary (e.g., voltage supplied to the GPU). Consequently, we use a TRNG implementation which is based on race conditions in multi-core environments caused by simultaneous memory accesses to shared variables. It takes advantage of uncertainties that arise when cores simultaneously access a particular memory location [40]. In our case, each simultaneous memory access unpredictably flips bits stored in shared variables. This unpredictability enables the GPU to generate noise which can be sampled and then used as an entropy source. We evaluated our implementation using statistical tests such as NIST SP 800-22 [36], DIEHARD [17], and ENT [47]. The TRNG implementation passes all standard tests and achieves a throughput of 4 kB/s on NVIDIA A100 GPUs and thus takes around 8 ms to generate an output of 256 bits. The TRNG provides 7.999 996 bits of entropy per byte (measured using ENT [47]).

7 Evaluation

Evaluation setup. To evaluate the performance of the checksum function, we use a setup based on an ASUS RS720-E10-RS12E equipped with a A100-PCIE-40GB GPU and Intel Xeon Gold 6348 CPU [12] which natively supports SGX instructions. We run the SGX enclave in both native and simulation mode. To benchmark the execution time of the verification process and evaluate runtime overheads, we also run the VF on a dual-socket system with an A100-SXM4-40GB and AMD EPYC 7742 CPU.

Register consumption. For the execution of the checksum function, the loop counter, data pointer, and the checksum result are stored in registers. In addition to those registers, we use 22 additional registers to store intermediate state during the computation of the checksum. In total, the checksum function verifies 524,288 bytes. The beginning of the buffer contains the checksum function itself, whereas the remainder is filled with pseudo-randomly generated values.

Experiment Nr.	1	2	3	4
self-modifying code	✗	✗	✓	✓
instructions	428	429	8,342	8,342
iterations	100,000	100,000	1,000	1,000
inner iterations	0	0	0	5000
inner instructions	0	0	0	216
verification (AMD) [s]	21.6	21.6	9.99	497
verification (Intel) [s]	102	102	47.0	2337
runtime T_{avg} [s]	0.4941	0.4977	0.1309	12.40
% of GPU peak perf.	99	98	75	100
adversarial NOP	✗	✓	✗	✗
runtime σ [s]	0.0009	—	—	—
runtime T_{min} [s]	—	0.4966	—	—
$T_{avg} + 2.5\sigma$ [s]	0.4964	—	—	—

Table 1: Evaluation of checksum implementations.

Summary of results. Table 1 summarizes our experiment series conducted to evaluate the performance of SAGE's VF. We distinguish between two categories depending on whether the checksum function contains self-modifying code or not. Depending on the category, the total number of instructions and number of checksum loop iterations are adapted. For each experiment, we report the VF's execution time on the GPU, the utilization ratio during the checksum execution, the verification time on the CPU, detection threshold, etc.

Experiment 1 demonstrates our best reference implementation. Experiment 2 simulates an attack on the checksum function from the first experiment. In Experiment 3, we show the effect on the performance of adding self-modifying code to the reference implementation. Experiment 4 shows a possible technique to compensate for the loss of performance with enabled self-modifying code.

7.1 VF Performance

To evaluate the performance of VF, we report its average runtime and utilization ratio during the checksum execution (Table 1). As a reference for this ratio, we use the *peak GPU performance*, which assumes that the number of warps that are executed concurrently per clock cycle is 4 (see §6.4).

We compare our reference implementation from Experiment 1 (in SASS) with the same code written in PTX (virtual assembly), that has been processed using the NVIDIA PTXAS assembler with the highest possible level of optimization enabled. In comparison, the optimized version of the checksum function that we generated using our instruction generation framework is around ∼230% faster than an implementation in PTX.

The checksum functions in Experiments 3 and 4 contain self-modifying code. This requires triggering cache eviction of the instruction cache such that the modified instruction gets updated. To trigger the cache eviction for the L2 instruction cache (128 kB), the checksum loop is required to be larger than the cache size. As a consequence, we use 8342 16 B instructions in the checksum loop. With this cache eviction strategy, our implementation is able to achieve 75% of the maximum utilization. Upon closer inspection with a GPU profiler, we find that 99% of all pipeline stalls that happen during the execution of the checksum function are caused by the fact that no instructions are available in the instruction cache to be executed. On average, each warp of this kernel spends 14.1 cycles being stalled due to not having the next instruction fetched yet. In comparison, reducing the size of the checksum loop to 6.7 kB (as in Experiment 1), we achieve a utilization of 99% without triggering cache eviction. This means that the hardware is unable to load the modified instructions in time for execution without causing any pipeline stalls. By comparing the VF's performance in Experiments 1 and 3, we can conclude that a higher utilization can be achieved in case other cache eviction strategies become available (see §6.4).

In addition to the previous experiment, we modified the checksum function by adding an "inner" loop to the main loop of the checksum function calculation (Experiment 4). This effectively hides the performance loss due to cache misses in the instruction cache and achieves 100% of the GPU peak performance. However, the time required to verify the code outside of the nested loop drastically increases and is thus considered too long to be practical.

7.2 Attack Robustness

To evaluate the robustness of our VF implementation with regards to attacks, we estimate the number of instructions that can be injected by an adversary without causing a noticeable time overhead. For this purpose, we measure the performance of the checksum function for 100,000 iterations and record the standard deviation σ of the total execution time based on 100 runs. We assume that the results of this experiment series are normally distributed and set the threshold value to detect adversarial tampering to be at $2.5 \cdot \sigma$ from the mean. The probability of a false positive is about 0.5%, in which case the verification process is restarted. Depending on the application requirements, the probability of false positives can be decreased at the expense of a larger number of iterations.

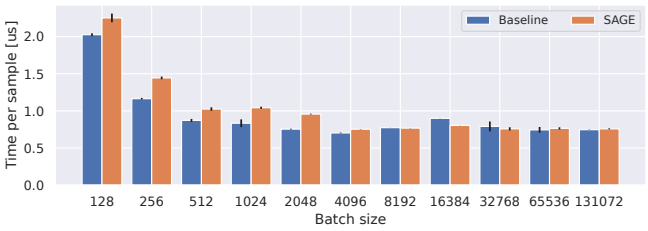

Figure 6: Time per batch sample in MLP.

To evaluate the robustness of this approach, we insert one additional NOP instruction in Experiment 2 (adversarial NOP) and report the minimum run time T_{min} (averaged over 100 runs). Assuming a detection threshold of $T_{avg} + 2.5\sigma$, we can conclude that $T_{avg} + 2.5\sigma < T_{min}$ and thus it is impossible to insert one or more instruction without detectable overhead.

Time measurement occurs on the CPU, including the time of communication between the CPU and GPU, in addition to the checksum computation loop. This raises the question of performance portability across hardware configurations. Since the configuration is fixed and assumed to be trusted, communication adds a constant time that can be measured offline once and then reused. This approach allows us to consider only the checksum loop in the detection threshold.

7.3 Memory Region Inclusion Probability

To evaluate how resilient our approach is regarding minor modifications in memory region containing the VF code (e.g., bit flips), we estimate the probability that a particular location is never included into the checksum result. We assume that memory accesses are distributed uniformly. Each block contains a single random memory access that loads an aligned 32-bit integer. For 2,500,000 iterations and a total checksum size of 524,288 integers, the probability that a memory location is never included in the checksum result is negligible:

$$(1 - 1/524288)^{2500000} = 0.0085$$

7.4 Runtime Overheads

Figure 6 shows a performance evaluation of SAGE using a multilayer perceptron (MLP) as an example. It consists of two Linear layers with weights of size `784×100` and `100×10`, with a Relu layer in between. As the workload increases, the overhead associated with the non-standard SAGE communication protocol becomes less noticeable.

The main sources of overhead are data transfers and kernel launches. Figure 7 shows that the copy operation requires additional time, which is a linear function of the input data size due to the additional data transfer between the host-accessible GPU memory and the GPU memory allocated by the SAGE kernel. SAGE adds less than 5% extra execution time to user

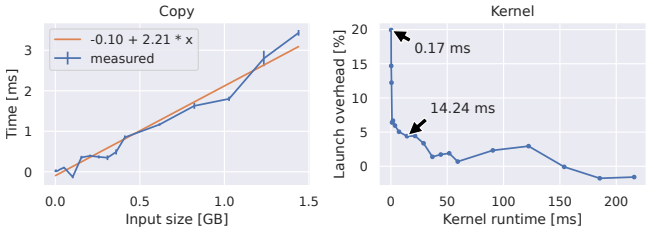

Figure 7: Overheads of data transfers and kernel launches.

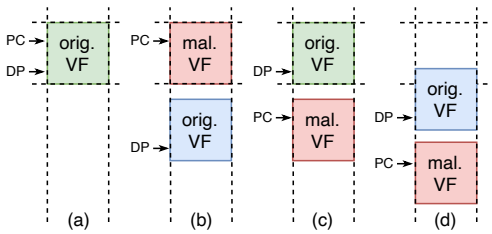

Figure 8: Memory copy attack variants.

kernels, which originally took more than 14.24 ms. Our evaluation does not take into account the overhead associated with the implementation of the encryption-decryption protocol, since its choice is left to the user.

7.5 Limitations of the Prototype

The use of self-modifying code requires triggering cache eviction of the instruction cache such that the modified instruction gets updated. With this cache eviction strategy, our implementation is able to achieve 75% of the maximum utilization. This is due to the GPU hardware not being able to load the required instructions in time for processing after the L2 cache eviction. If other cache eviction strategies become available to user code, higher utilization can be achieved. Unfortunately, triggering cache eviction using a large checksum loop limits the time difference caused by an adversary inserting instructions into the checksum loop. We believe that GPU vendors with in-depth knowledge of GPU architecture would be able to reduce the checksum loop size and use self-modification.

8 Security Analysis

In the following, we systematically analyze potential attacks given our threat model (see §3.3).

Pre-computation. The result of the checksum function depends on an unpredictable challenge issued by the verifier enclave. This prevents pre-computation attacks where the checksum value or part of the checksum (e.g., intermediate values) are pre-computed to later run code other than the VF.

Computation optimizations. The checksum function implementation must be time-optimal as algorithmic optimization would allow the adversary to find computationally faster or more efficient way of computing the checksum value (see §6.3 for details). SAGE is designed to prevent optimization, but to achieve provable guarantees the method of Gligor and Woo could be applied [8].

Attacks on the host system. The host system is untrusted (except for the verifier enclave) and the adversary is assumed to have administrative control over the system. This enables the adversary to eavesdrop, intercept, modify, or delay challenges or checksum results being transmitted between the verifier and the device. Given that the communication channel during the checksum computation is unauthenticated, the adversary could also inject challenges or checksum results. Modifications to the challenge would lead to a different checksum result. By injecting challenges the adversary could treat the VF as an oracle; however, given the unpredictable challenge generation, the probability of the verifier reusing the same challenge value is negligible.

Attacks on the device / Resource takeover. Before running the verification function, the device is considered untrusted. An adversary could be present on the device and interfere with the execution of the VF (e.g., by replacing or reordering instructions). This is prevented by the self-verification property and the strongly-ordered design of the checksum function. A strongly-ordered function requires the adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result. Otherwise, the output differs with high probability if operations that have dependencies among them are evaluated in a different order.

Our design uses all available SMs simultaneously and maximizes thread and register usage. Thus, if an adversary would run a computation, the checksum computation would be deferred resulting in a considerable time overhead. However, the execution of a user kernel might not require all available GPU resources and would allow the adversary to take over these available resources. To prevent such attacks we require the user to develop kernels to achieve maximum GPU occupancy.

Memory copy attacks. Seshadri et al. [32,33] specify memory copy attacks that can be conducted by the adversary in the following three different ways as illustrated in Figure 8:

(b) The adversary replaces the checksum function with an altered checksum function and executes it, but computes the checksum over a correct copy of the checksum function elsewhere in memory. Thus, the program counter is correct, but the data pointer points to the original copy of the checksum function in a different memory location.

(c) The adversary uses the correct checksum function code in the original memory location to compute the checksum value, but executes a modified checksum function elsewhere in memory. Thus, the data pointer points to the original checksum function, but the program counter will be different.

(d) The adversary places both the original checksum function code and its altered version elsewhere from the memory locations where the correct checksum code originally resided. Thus, both the program counter and the data pointer will be different compared to an execution of the original checksum function.

To prevent memory copy attacks, both the program counter and the data pointer need to be included in the computation of the checksum. The DP is included in each step of the

computation, whereas the PC is indirectly included using self-modifying code. In addition to these specified attacks, the attacker could also copy the entire checksum function to a different location. This will not lead to a successful attack because the absolute location of the checksum function is irrelevant to security as long as the function pointer and the data pointer remain the same relative to the location of the effective checksum function in memory.

Proxy attacks. We refer to proxy attacks as attacks where the adversary eavesdrops on the communication and obtains the challenge sent to the device, sends it to a proxy, computes the checksum function there and returns the result to the verifier. We distinguish between the following cases. *GPUs on the same host*: we establish (and maintain) root-of-trust in sequence starting from the most powerful GPU to the least powerful one. *GPUs on a different host*: by involving a remote entity, the measured execution time will increase by the network latency for both sending the challenge and receiving the response. Tuning the number of checksum iterations to make the detection threshold smaller than the network latency, prevents using a more powerful GPU in a remote location.

Time-of-check to time-of-use attacks [5] / Execution environment takeover. In SAGE, these attacks are considered because the checksum computation happens prior to the execution of the user kernel. In particular, the adversary has two points where it could take over the execution environment set up by the VF: 1) before the launch of the user kernel, and 2) after the execution of the user kernel has completed. The former case is prevented by launching the user kernel(s) from within the VF epilogue. In the latter case, the execution of the user computation has finished and thus the user is indifferent whether the dynamic root-of-trust has been compromised. If the user wants to execute another kernel, the dynamic RoT needs to be re-established.

Replay attacks. To protect against duplicate transmissions of encrypted code and data between the SGX enclave and GPU, we add sequence numbers to each transmission.

Execution integrity and memory protection. While code and data are encrypted in transit, they have to be decrypted before use and placed into GPU memory. We control the allocation of memory from the kernel caller by calling `malloc()` from the device code. CUDA guarantees that the memory allocated in this way (unlike `cudaMalloc()`) is not accessible from the CUDA runtime or driver API. Therefore, even an attacker with root access to the operating system will not be able to access application code and data from such areas.

8.1 Formal Verification of Modified SAKE

To show that our modified SAKE protocol securely establishes a key between the verifier and the GPU, we have formally modeled the key establishment protocol and verified its security properties using the Tamarin prover [38] under the assumption that the computed checksum provides a short-lived secret. To model this property in Tamarin, we use a single-use authentic channel over which we send $w_2, \mathrm{MAC}_c(w_2)$. We show that the established symmetric key remains secret and is unique, a weak agreement exists between the verifier and the device, and recent liveness for each run of the protocol [37].

9 Related Work

To support trusted execution on GPUs, the following approaches were proposed. Graviton [46] specifies an architecture for supporting trusted execution environments on GPUs by changing the GPU's command processor to perform remote attestation based on device specific keys and ensure isolation between multiple processes running on the GPU. This is achieved by utilizing a set of keys where the root key gets embedded into the hardware of the device upon its creation. The latter requires modification to the GPU hardware by modifying the GPU's internal command processor to impose a strict ownership discipline.

HIX [13] proposes a heterogeneous isolated execution environment. HIX does not require modifications to the GPU architecture to offer an isolated execution environment, but instead physically modifies the I/O interconnect between the CPU and GPU and refactors the GPU device driver to work from within a TEE on the host. The TEE can then allocate trusted enclaves on the GPU.

HETEE [49] is based on a standalone computing system to dynamically allocate accelerators (such as GPUs or FPGAs) for either secure computing, or available to the host OS using PCIe switches. The security controller (and its software) is assumed to be trusted and interacts with the management CPU to control PCIe switching. HETEE attempts to provide isolation by selectively making accelerators available to specific applications by controlling communication to the accelerator through the security controller.

Telekine [9] illustrates side-channel attacks against TEE on GPUs based on observing the timing of GPU kernel execution. It then introduces a GPU stream abstraction that ensures execution and interaction through untrusted components are independent of any secret data. Telekine requires a GPU TEE to be deployed.

Machine learning represents a major use case for using GPUs as accelerators and can require privacy-preserving approaches for sensitive data. Slalom [42] uses a combination of a trusted enclave and untrusted GPU. The system decomposes the machine learning into two parts, where the control flow part runs inside the trusted enclave and operations that are not privacy sensitive (such as convolutions based on matrix multiplications) are offloaded to the GPU. Unfortunately, the split results in a decrease of training and inference accuracy.

SOTER [35] relies on the associativity property of operators present in DNN models. It assumes that the GPU is untrusted and sends modified parameter data from the SGX enclave. The output data received from the GPU is converted again to produce the expected result. To check the integrity of

the result, SOTER creates challenges and expects the GPU to return a proof of computation. If the integrity is compromised, the proof will be incorrect.

9.1 Hardware-based Attestation

NVIDIA has introduced a *confidential computing* [26] feature in the Hopper architecture. While technical details are currently scarce, this feature is touted to require no changes to the application code, while ensuring both the confidentiality and integrity of data and code running on the device.

With the addition of a vendor-backed hardware TEE solution, there is a question regarding the relevance of software-based attestation. While a hardware implementation provides reasonable levels of security, there have been several examples where the hardware-based techniques were flawed [6, 28, 44]. In these cases software-based techniques could come to the rescue. Software attestation can be complementary to the newly-added confidential computing feature and add another level of security to achieve defense in depth. Further, since the software layer doesn't rely on the private keys embedded in hardware by the manufacturers, it also reduces the TCB.

The trust required to obtain the properties provided by attestation is further reduced by combining both hardware- and software-based approaches. In essence, as long as one of the attestation methods is secure, the properties obtained using attestation hold.

9.2 Software-based Attestation

SWATT [33] uses a verification function that is based on pseudo-random memory traversal to compute the checksum. The verifier measures the execution time and verifies the checksum. Malicious code is required to verify each memory access to replace memory reads of changed locations with expected content, resulting in detectable time overhead. SWATT checks the entire memory of a system and its running time becomes prohibitive on systems with large memories.

PIONEER [32] verifies the integrity and guarantees the execution of code using a checksum function that is closely tied to the Pentium 4 architecture. The checksum function computes a fingerprint of the verification function and sets up an untampered execution environment. It is constructed such that manipulations by the adversary will noticeably increase the computation time.

Kovah et al. [16] and Butterworth et al. [3] extended the checksum computation to work on a Microsoft Windows system (CPU only), enabling a remote verifier to attest to a running system in a corporate environment.

Shaneck et al. [34] describe a software-based approach to remotely attest the static memory contents of sensors without requiring any additional hardware on the sensors nor precise measurements of execution timing. They use self-modifying code that generates memory read and jump instructions during the execution of their code.

Gligor and Woo [8] proposed a system that allows to provably establish a root of trust and provide secure initial states for all software unconditionally. The authors design a family of k-independent (almost) universal hash functions based on polynomials and use Horner's rule to show time- and memory-optimal evaluation of polynomials. An interesting area of future work is to translate these results to the context of computation on GPUs.

10 Conclusion

The prospect of software-only trust root establishment and secure code execution on GPUs offers exciting opportunities: execution of sensitive GPU code that should not be leaked to the GPU operator (code secrecy), correct execution of GPU code in an adversarial environment (code and execution integrity), preserving data correctness and confidentiality in the presence of malicious code on the system (data secrecy and integrity). SAGE represents a first step for achieving these properties on the NVIDIA Ampere architecture, under the circumstances that the architectural details about the Ampere architecture are closed-source. Since architectural knowledge for designing the verification function (VF) is key, our software-based approach to provide secure code execution on GPU paves the way forward for GPU vendors: they are naturally in a position to align the design of the VF to their architectural knowledge and lead the standardization process for trust establishment on GPUs.

Remaining open challenges include the design of software-based secure execution on alternative platforms, improving the execution speed of the verification function, and extend the execution model to support libraries that use a hybrid CPU+GPU compute model (e.g., TensorFlow [41]). Ultimately, an interesting future research question to answer is the interplay between hardware- and software-based approaches for trusted execution to achieve the strongest possible security properties for GPU-based execution.

Acknowledgments

This work was supported by the European Union's HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION). We thank CSCS for providing access to compute resources used for this work.

References

[1] Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *ACM SIGOPS Operating Systems Review*, 32(4):9–20, 1998.

[2] ARM. ARM TrustZone technology. https://developer.arm.com/ip-products/security-ip/trustzone, 2021. [Online; accessed 01-Feb-2022].

[3] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, November 2013.

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

[5] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the TOCTOU problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[6] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in Intel Management Engine. *Black Hat Europe*, 2017.

[7] Bruno Forlin, Ronaldo Husemann, Luigi Carro, Cezar Reinbrecht, Said Hamdioui, and Mottaqiallah Taouil. G-PUF: An intrinsic PUF based on GPU error signatures. In *IEEE European Test Symposium (ETS)*, pages 1–2, 2020.

[8] Virgil D Gligor and Shan Leung Maverick Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[9] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 817–833, 2020.

[10] Intel. Interface for generic crypto library APIs required in SDK implementation. https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_tcrypto.h, 2021. [Online; accessed 01-Feb-2022].

[11] Intel. Software Guard Extensions for Linux. https://github.com/intel/linux-sgx, 2021. [Online; accessed 01-Feb-2022].

[12] Intel. Xeon Gold 6348 Processor. https://ark.intel.com/content/www/us/en/ark/products/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz.html, 2021. [Online; accessed 01-Feb-2022].

[13] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–468, 2019.

[14] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[15] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[16] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New results for timing-based attestation. In *IEEE Symposium on Security and Privacy (SP)*, May 2012.

[17] George Marsaglia. DIEHARD: A battery of tests of randomness. 1996.

[18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe performance for end host networking. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[19] NVIDIA. Ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/, 2020. [Online; accessed 01-Feb-2022].

[20] NVIDIA. How NVIDIA EGX is forming central nervous system of global industries. https://blogs.nvidia.com/blog/2020/05/15/egx-security-resiliency/, 2020. [Online; accessed 01-Feb-2022].

[21] NVIDIA. CUDA binary utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, 2021. [Online; accessed 01-Feb-2022].

[22] NVIDIA. CUDA occupancy calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html, 2021. [Online; accessed 01-Feb-2022].

[23] NVIDIA. INT 32 and FP64 can be used concurrently in the Volta architecture? https://forums.developer.nvidia.com/t/int-32-and-fp64-can-be-used-concurrently-in-the-volta-architecture/108729/4, 2021. [Online; accessed 01-Feb-2022].

[24] NVIDIA. Multi-instance GPU user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2021. [Online; accessed 01-Feb-2022].

[25] NVIDIA. Basic linear algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas, 2022. [Online; accessed 01-Feb-2022].

[26] NVIDIA. NVIDIA H100 Tensor Core GPU architecture. https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper, 2022. [Online; accessed 12-Dec-2022].

[27] Lena E Olson, Jason Power, Mark D Hill, and David A Wood. Border control: Sandboxing accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 470–481. IEEE, 2015.

[28] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.

[29] Research and Markets. Data center accelerator market – global forecast to 2026. https://www.researchandmarkets.com/reports/5390148/data-center-accelerator-market-by-processor-type, 2021. [Online; accessed 01-Feb-2022].

[30] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. Decay-based DRAM PUFs in commodity devices. *IEEE Transactions on Dependable and Secure Computing*, 16(3):462–475, 2019.

[31] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software attestation for key establishment in sensor networks. In *International Conference on Distributed Computing in Sensor Systems*, pages 372–385, 2008.

[32] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM symposium on Operating systems principles*, pages 1–16, 2005.

[33] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy (SP)*, pages 272–282, 2004.

[34] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In *European Workshop on Security in Ad-hoc and Sensor Networks*, pages 27–41. Springer, 2005.

[35] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, Fengwei Zhang, and Heming Cui. SOTER: Guarding black-box inference for general neural networks at the edge. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 723–738, Carlsbad, CA, July 2022. USENIX Association.

[36] Elaine Barker Smid, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication (NIST SP), National Institute of Standards and Technology. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=906762, 2010. [Online; accessed 01-Feb-2022].

[37] Tamarin Team. Tamarin manual - property specification. https://tamarin-prover.github.io/manual/book/007_property-specification.html, 2021. [Online; accessed 01-Feb-2022].

[38] Tamarin Team. Tamarin prover. https://tamarin-prover.github.io/, 2021. [Online; accessed 01-Feb-2022].

[39] TechSpot. Intel's SGX deprecation impacts DRM and Ultra HD Blu-ray support. https://www.techspot.com/news/93006-intel-sgx-deprecation-impacts-drm-ultra-hd-blu.html, 2022. [Online; accessed 01-Feb-2022].

[40] Je Sen Teh, Azman Samsudin, Mishal Al-Mazrooie, and Amir Akhavan. GPUs and chaos: A new true random number generator. *Nonlinear Dynamics*, 82(4):1913–1922, 2015.

[41] Tensorflow. An end-to-end open source machine learning platform. https://www.tensorflow.org/, 2021. [Online; accessed 01-Feb-2022].

[42] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[43] Pol Van Aubel, Daniel J Bernstein, and Ruben Niederhagen. Investigating SRAM PUFs in large CPUs and GPUs. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 228–247, 2015.

[44] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural Load Value Injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72, 2020.

[45] Sebastiano Vigna. Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, 315:175–181, 2017.

[46] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 681–696, 2018.

[47] John Walker. ENT – A pseudorandom number sequence test program. https://www.fourmilab.ch/random/, 2008. [Online; accessed 01-Feb-2022].

[48] Jun Zhao, Virgil Gligor, Adrian Perrig, and James Newsome. ReDABLS: Revisiting device attestation with bounded leakage of secrets. In *Cambridge International Workshop on Security Protocols*, pages 94–114, 2013.

[49] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Lutan Zhao, Fengkai Yuan, Peinan Li, Zhongpu Wang, Boyan Zhao, et al. Enabling privacy-preserving, compute-and data-intensive computing using heterogeneous trusted execution environment. *arXiv preprint arXiv:1904.04782*, 2019.