GRIM: Leveraging GPUs for Kernel Integrity Monitoring

Lazaros Koromilas[1]* Giorgos Vasiliadis[2]
Elias Athanasopoulos[3] Sotiris Ioannidis[4]

[1] koromilaz@gmail.com

[2] Qatar Computing Research Institute, HBKU
gvasileiadis@qf.org.qa

[3] Vrije Universiteit Amsterdam
i.a.athanasopoulos@vu.nl

[4] FORTH, Greece
sotiris@ics.forth.gr

Abstract. Kernel rootkits can exploit an operating system and enable future accessibility and control, despite all recent advances in software protection. A promising defense mechanism against rootkits is Kernel Integrity Monitor (KIM) systems, which inspect the kernel text and data to discover any malicious changes. A KIM can be implemented either in software, using a hypervisor, or using extra hardware. The latter option is more attractive due to better performance and higher security, since the monitor is isolated from the potentially vulnerable host. To remain under the radar and avoid detection it is paramount for a rootkit to conceal its malicious activities. In order to detect self-hiding rootkits researchers have proposed *snooping* for inferring suspicious behaviour in kernel memory. This is accomplished by constantly monitoring all memory accesses on the bus and not the actual memory area where the kernel is mapped.

In this paper, we present GRIM, an external memory monitor that is built on commodity, off-the-shelf, graphics hardware, and is able to verify OS kernel integrity at a speed that outperforms all so-far published snapshot-based systems. GRIM allows for checking eight thousand 64-bit values simultaneously at a 10 KHz snapshot frequency, which is sufficient to accurately detect a self-hiding loadable kernel module insertion. According to the state-of-the-art, this detection can *only* happen using a snoop-based monitor. GRIM does not only demonstrate that snapshot-based monitors can be significantly improved, but it additionally offers a fully programmable platform that can be instantly deployed without requiring any modifications to the host it protects. Notice that *all* snoop-based monitors require substantial changes at the microprocessor level.

* This work was performed while at FORTH, Greece.

1 Introduction

Despite the recent advances in software security, vulnerabilities can still be exploited if the adversary is really determined. No matter the protection enabled, there is always a path for successful exploitation, although admittedly, today, following this path is much harder than it was in the past. Since securing software is still under ongoing research, the community has investigated alternative methods for *protecting* software. One of the most promising is monitoring the Operating System (OS) for possible exploitation. Once an abnormality is detected then the monitor should be able to terminate the system's operation and alert the administrator.

This form of protection is commonly offered by tools known as *Kernel Integrity Monitors* (KIMs). The core operation of these tools is to inspect, as frequently as possible, both the kernel code and data for determining if something has been illegally modified. In principle, compromising an operating system is usually carried out using a kernel rootkit (i.e., a piece of malicious code that is installed in the OS), which usually subverts the legitimate operation of the system by injecting malicious functionality. For example, the simplest way for achieving this is by inserting a *new* (malicious) system call, which, obviously alters a fundamental structure in the kernel's code: the system-call table. In order to identify such a simple rootkit, it is enough to only monitor the memory region where the system-call table is mapped for possible changes.

Implementing KIMs may sound trivial, however the level of sophistication of modern kernel rootkits, gives space for many different choices. A straightforward approach is to implement the monitor solely in software, in the form of a hypervisor which runs and frequently introspects the OS for possible (malicious) changes [6, 10, 25, 29]. This choice is really convenient, since there is no need for installing custom hardware, nevertheless it is implied that the monitor's code is non vulnerable itself. Unfortunately, it has been demonstrated that a hypervisor can be compromised by code running at the guest OS [24]. In addition, formally verifying the monitor's code may need significant effort [17]. A viable alternative is to offer monitors that are implemented in hardware. Copilot [13] is a representative architecture, implemented in a PCI card and it is basically a snapshot-based monitor. Essentially, a snapshot-based system monitors a particular memory region to identify possible malicious changes. As an example, consider the simple case of monitoring the region where the system-call table has been mapped, in order to detect if a new (malicious) system call has be injected. Copilot has a transparent operation, allowing the OS to be unaware of its existence, and thus it stands as a very attractive option, especially in terms of deployment. Still, modern rootkits have evolved and developed techniques that can evade detection, by exploiting the window of opportunity between two snapshots. As a matter of fact, a rootkit can simply perform a (malicious) change right after a snapshot is taken and subsequently remove it before the next snapshot is available.

To overcome this inherent limitation of snaphost-based detection systems, recent proposals have been focused on *snooping* based detection [18, 22]. A snoop-based system monitors all the operations that are sent over the memory bus. In the context of the aforementioned example we used, the snoop-based detector would have achieved equivalent detection with the snapshot-based system by capturing the *write* operations that aim at modifying the region where the system-call table is mapped. It is evident, that the snoop-based approach performs a lighter check, since instead of monitoring a particular region, it monitors the bus for a particular operation. Nevertheless, snooping is possible *only* in custom processors, since the memory controller is integrated to the CPU, which poses critical deployment issues. The benefits of snoop-based systems were introduced by Vigilare [22] and have been demonstrated in KI-Mon [18] where, in short, the authors provide experimental evidence that a snapshot-based approach can only reach 70% of detection rate, while their snoop-based system, KI-Mon, can reach 100% of detection rate.

In this paper, we acknowledge the benefit of snoop-based systems, such as KI-Mon, but we stress that snapshot-based systems can essentially *do better*. We implement GRIM, a novel snapshot-based KIM based on a GPU architecture. Using GRIM we can easily reach 100% of detection rate using a snapshot-based *only* architecture. GRIM does not aim to justify that excellent engineering can simply optimize a system. Instead, in this paper we promote the design of a novel architecture, which *does not only* demonstrate high detection rates in snapshot-based integrity monitors, but, also, provides a generic extensible platform for developing KIMs that can be instantly deployed. GRIM works transparently and requires no modifications such as re-compilation of the kernel and installing custom hardware on the system it protects. In addition, GRIM does not aim at simply promoting the usage of GPUs in a new domain. To the contrary, GRIM delivers a design that demonstrates many critical properties for KIMs. Beyond the dramatic speed gains in detection rates, the system is easily programmable and extensible, while it is not based on custom hardware but on commodity devices.

To summarize, we make the following contributions:

- We design, implement, and evaluate GRIM, a novel GPU-based kernel integrity monitor.
- GRIM demonstrates that snapshot-based monitors can do substantially better than it has been so far documented in current literature [18]. We are able to reach easily 100% detection rate, surpassing substantially the reported detection rate (70%) in the state of the art.
- GRIM is fully programmable and it provides a generic extensible platform for developing KIMs that can be instantly deployed using just commodity devices. It works transparently and requires no modifications such as re-compilation of the kernel and installing custom hardware on the system it protects.

2 Background

In this section, we describe the architecture of modern graphics cards, with an emphasis on their general-purpose computing functionalities that provide for non-graphics applications.

2.1 GPUs and CPUs

Graphics cards nowadays have significant more computing resources than they used to have a couple of decades ago. The processing speeds they achieve and their low cost makes them a good candidate for many applications beyond graphics rendering [26, 27]. They contain hundreds of processor cores, which can be programmed by general-purpose programming frameworks such as OpenCL [3] and CUDA [23], and thus transformed to general-purpose computing platforms.

In general, GPUs execute programs organized in units called *kernels*. The main difference with programs that run on a CPU is that the execution model is optimized for data-parallel execution. The majority of its chip area is devoted to computation units, rather than data caching and flow control. As a result, maximum gains on GPUs are achieved when the same instructions run on all threads concurrently. In contrast, CPUs have big caches accounting for half of the chip and large control logic, since their main target is optimizing a single thread.

2.2 The GPU memory hierarchy

The NVIDIA CUDA architecture, which is used for the development of the prototype presented in this paper, offers different memory spaces, optimized for different usages. The host allocates memory for GPU kernels in *global* memory space which maps to off-chip DRAM memory. Furthermore, the *constant* memory space is optimized for read-only accesses, while the *texture* memory space for 2D spatial locality. Allocations to all these types of memory are visible to the host and persistent across kernel launches in the same process context. Individual threads have access to *local* memory, which could reside in global memory though, and is dedicated to a single thread as local storage for automatic variables. The GPU cores, called streaming processors (SP), are grouped together in multiprocessors. Depending on the device generation/model (CUDA compute capability) all different types of global (constant, texture, local) memory accesses are cached in possibly separate caches in each multiprocessor.

Threads are organized in blocks and each block runs on cores of the same multiprocessor. The *shared* memory space offers fast access to the whole thread block. The absolute sizes differ across architectures but some typical sizes are the following. Shared memory is 64 KB per group of 32 threads (warp) inside the thread block, but some of it is also used as a first-level cache by the hardware itself. Registers are not fully addressable, so they are not accessible from the host side. Values can, however, spill to global memory due to excessive allocation [23] which also makes accesses slower.

2.3 GPUs for Kernel Integrity Monitoring

In order to monitor the integrity of host memory, a coprocessor-based system, like GRIM, must meet, at a minimum, the following set of requirements [13]:

- **Independence.** GRIM has to operate in complete isolation, using a single dedicated GPU, and must not rely on the host for monitoring the system's integrity. The GPU must be used exclusively for memory monitoring, hence it cannot be used for other purposes. Any extra GPUs can be easily appended to serve other usages, if necessary, without affecting the proper usage of the kernel monitor. Moreover, GRIM must continue to operate correctly and report all malicious actions, regardless of the running state of the host machine, especially when it has been compromised.
- **Host-memory access.** GRIM must be able to access the physical memory of the host directly for periodically checking its integrity and detecting any suspicious or malicious actions.
- **Sufficient computational and memory resources.** GRIM must contain enough memory to keep a baseline of system state. Moreover, it must have sufficient on-chip memory that can be used for private calculations and ensure that secret data would not be leaked or held by an adversary that has compromised the protected system. In addition, GRIM should be able to process large amounts of data efficiently and perform any operation requested.
- **Out-of-band reporting.** GRIM must be able to report the state of the host system in a secure way. To do so, it needs to establish a secure communication channel and exchange valid reports, even in the case the host has been fully compromised.

In order to meet the above requirements, several characteristics of the GPU's execution model require careful consideration. For instance, GPU kernels typically run for a while, perform some computation and then terminate. While running, a GPU kernel can be terminated by the host or swapped with another one. This model is not secure, as the coprocessor needs to execute in isolation, without being influenced by the host it protects. Essentially we stress that leveraging GPUs for designing an independent environment with unrestricted memory access that will monitor the host's memory securely, is not straight forward, but rather challenging. Many GPU characteristics must be considered carefully and in a particular way. In the following sections we describe how we implement and enforce these requirements in a real system.

2.4 The GPU execution model

Execution on the GPU involves two sides, namely the host and the device. The host prepares the device and then signals execution. The basic steps are: *(i)* copying the compiled kernel code to the device, *(ii)* transferring the input buffers to device memory via DMA, *(iii)* running the kernel on the device, *(iv)* transferring the output buffers back to host memory via DMA.

The NVIDIA architectural details are not public but there is substantial work available on reversing the runtime and device drivers [14, 21]. From what we already know, there is a host-side memory-mapped buffer that is used by the driver to control the device. API calls for data transfers and program execution translate to commands through this driver interface. Finally, there are alternative runtime libraries and drivers that offer CUDA support such as Gdev [5], Nouveau [1] and PSCNV [4].

2.5 Threat Model

We assume that the adversary has the capability to exploit vulnerabilities in any software running in the machine after bootup. This includes the OS and all of its privileged components. We also assume that the adversary does not have access to the hardware, and thus cannot replace the installed GPU with a malicious one using the same driver interface.

In-Scope Threats. Snapshot-based kernel integrity monitor techniques aim to ensure the integrity of the operating system of the already compromised host, and primarily to detect modifications on memory regions that are considered immutable after boot, such as the text of the kernel and any of the loaded LKMs, as well as the contents of their critical data structures.

Out-of-Scope Threats. Sophisticated rootkits [7,11,12,28] that evade snapshot-based kernel integrity monitors are out of scope. For example, there are CPU-controlled address translation methods that can be used to mount address space relocation attacks, by changing the page directory pointer of the kernel context [12]. So far, there is an arms race between building techniques that allow a rootkit to evade detection and bypass a KIM, and building detection methods that are able to capture these highly sophisticated attacks. To this aspect we contribute a new architectural paradigm for building fast snapshot-based KIMs that can potentially integrate the state-of-the-art detection algorithms.

3 Design

In this section we describe the design of GRIM at the hardware and software level, and we show how we can leverage modern GPUs as a monitoring mechanism that meets the requirements described in section 2.3.

GRIM is an external, snapshot-based, integrity monitor, that can also provide programmability and easy deployment. The overall architecture is shown in Figure 1. Essentially, the GPU reads the specified kernel memory regions over the PCI Express bus, via DMA. Each region is investigated in terms of integrity, and any abnormal or suspicious status is reported to an external admin station that is connected on the local network.

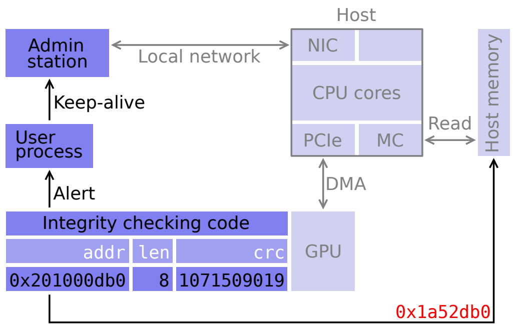

Fig. 1: Hardware/software architecture highlighting the monitor access path. The GPU is configured to periodically check a list of host memory regions against known checksums by reading the corresponding device virtual addresses (left). The user program periodically sends an encrypted sequence number together with a status code to a separate admin station. This mechanism defends against man-in-the-middle and replay attacks on the reporting channel.

From a software perspective, GRIM has two counterparts that run concurrently: the device program (GPU code) and the host program (user process). The device program is responsible for checking the integrity of a requested list of memory regions, and raise any alerts. The host program periodically reads the status area and forwards it to the admin station in the form of a keep-alive message. The only trusted component is hosted on the GPU; the user process cannot be trusted, so there is an end-to-end encryption scheme between the GPU and the admin station to protect against attacks, as we explain in §3.

Autonomous GPU execution. GRIM is designed to monitor the operating system's integrity isolated from the host, which may be vulnerable and could be compromised. For that reason, code and data used by GRIM must not be tampered with by an adversary. Modern GPU chips follow a cooperatively scheduled, non-preemptive execution style. That means that only a single kernel can run on the device at any single point in time. As we describe later on in §4.1 and also illustrated by previous work [26], we employ a bootstrapping method that forbids any external interaction with GRIM. Any attempt to kill, pause, or suspend the GPU component of GRIM results in system shutdown (as we will describe in §3), and the system can only resume its operation by repeating the bootstrap process. Some NVIDIA models conditionally support concurrent kernel execution, but those conditions can be configured. Therefore, even when running on those models, GRIM's kernel occupies all resources and no other, possibly malicious, code can run concurrently with it.

While these procedures ensure that GRIM can run safely once it has been initialized, current GPU programming frameworks such as CUDA and OpenCL have not been designed with isolation and independence in mind. Some drivers would even kill a context by default if its program appears to be unresponsive [2]. We configure the device to ignore these type of checks. Also, by default, only one queue, or stream in CUDA terminology, is active and the host cannot issue any data transfers before the previous kernel execution is finished. This can be addressed by creating a second stream dedicated to the GPU kernel of GRIM. Therefore, all data communication with the host and the admin station is performed using a separate stream.

Host memory access. An important requirement for morphing the GPU into a kernel integrity monitor is to establish a mechanism to reference the corresponding memory pages that need to be monitored. Unfortunately, current GPGPU frameworks, such as CUDA and OpenCL, use a virtual address layer that is unified with the virtual memory of the host process that utilizes the GPU each time. Since GRIM must access the kernel's memory (and not userspace), the memory regions of the kernel that are monitored should be mapped to the user process.

Typically, modern OSes, including Linux and Windows, prohibit users to access memory regions that have not been assigned to them. An access to a page that is not mapped to a process' virtual address space is typically considered illegal, resulting in a segmentation violation. To access the memory regions where the OS kernel and data structures reside, the particular pages must be located and mapped to GRIM user-space (i.e., the host counterpart that runs on the CPU). This is needed as an intermediate step for finally mapping these regions to the GPU address space.

To overcome the protection added by the OS, we use a separate loadable kernel module that is able to selectively map memory regions to user-space. Then, we are able to register these memory regions to the device address space, through the CUDA programming API. Due to the fact that the GPU is a peripheral PCIe device, it only uses physical addressing to access the host memory. Hence, after the requested memory registration, the GPU is able to access the requested kernel memory regions directly, through the physical address space. This feature allows us to un-map the user-space mappings of the kernel memory regions during the bootstrap phase, that would otherwise pose significant security risks.

Integrity monitoring. The memory regions to be monitored are specified by the user, and can include pages that contain kernel or LKM text, as well as arrays that contain kernel function pointers (i.e., jump tables). Hashing the static text parts of the kernel or the already loaded LKMs is straightforward. However, the OS kernel is fairly dynamic. Besides the static parts, there are additional parts that change frequently; for example, the VFS layer's data structures change every time new filesystems are mounted or removed. Also, every loaded kernel module can add function pointers.

Given the general-purpose programmability of modern GPUs, it is possible to implement checks that would detect malicious events, by performing several, multi-step, checks on different memory regions. These multi-step checks can become complex in cases where several memory pointers need to be dereferenced in order to acquire the proper kernel memory address. To support this kind of checks we need to walk the kernel page table and resolve the pointer's virtual address dynamically from the GPU. Assuming that we can already access the parts of the page table needed to follow a specific virtual address down to a leaf page table entry (PTE), we end up with a physical page number.

Accessing any physical page is not an inherent limitation of a peripheral PCIe device, such as the GPU. Ideally, the GPU can perform host page table walks, by reading the corresponding physical pages, and dereferencing any virtual page directly. However, the closed-source nature of the CUDA runtime and driver, on which we have based our current design, restrict us from accessing the requested physical page, if the latter has not been registered at the bootstrap phase, via the specialized API function call. For the time being, we do not support dynamic page table resolutions, instead we provide a static list of kernel memory regions, resolve their mappings, and create GPU-side mappings before entering the monitor phase, as we explain in §4.1. We note, however, that this is not a limitation of our proposed architecture, as the development of open-source frameworks (e.g. Gdev [5]) and drivers (e.g. Nouveau [1], PSCNV [4], etc.) would make a one-to-one mapping of all physical pages in the GPU address space practical. Exploring mapping of additional physical pages in the GPU's address space at run-time is part of our future work.

Sufficient resources. Modern GPUs are equipped with ample memory (up to 12 GB) and hundreds (or even thousands) of cores. Having such a wide memory space, gives us the ability to store plenty of kernel-image snapshots and enough state for detecting complicated, multi-step, types of attacks. Obviously, these kind of checks can become quite complicated, mainly due to the lack of a generic language that will allow the modelling of such scenarios on top of our architecture. Even though we do not allow such sophisticated memory checks at the moment, the process of aggressively reading and hashing memory has been tested, and, as we show in §5, the GPU prevails the resources to support this. Implementing sophisticated attacks against GRIM and evaluating the system's effectiveness against them is part of our future work.

Out-of-band execution. In the context of GRIM, the GPU acts as a coprocessor with limited defenses against itself. For example, an adversary that has compromised the host system could easily disable or reset the GPU device, and block any potential defensive actions. To overcome this we deploy a completely separate admin station that is responsible for keeping track of the host's proper state.

In particular, the user program that is associated with the GPU context, periodically sends keep-alive messages to the admin station through a private connection. Obviously, simply sending a message to raise an alert can be unsafe, because it is hard for the admin station to distinguish normal operation from a network partition or other failure. Therefore, we use keep-alive messages that encapsulate a GPU-generated status. These messages are encrypted together with a sequence number, to prevent an attacker from imitating GRIM and send spoofed keep-alive messages or replay older ones. Subsequently, the admin station is involved in the bootstrapping process because the secure communication channel with the host is established at that point. The exact communication protocol is described in §4.

On the admin station, a program logs the reports and makes sure that the monitor is always responsive. The admin station is responsible to take any specified action, every time a message that contains an alert is received or in error cases. An error case can be an invalid message or a missed packet (initiated by a time-out).

4 Implementation

In this section we provide implementation details and discuss technical issues we encountered in the development of GRIM. The current prototype is built on the NVIDIA CUDA architecture, and is portable across all CUDA-enabled NVIDIA models.

4.1 Mapping kernel memory to GPU

During bootstrapping, GRIM needs to acquire the kernel memory regions that need to be monitored. These regions are located in the kernel virtual address space. Therefore, the first step is to map them to the address space the GPU driver requires them to live, which is the virtual address space of the user process that issues the execution of the kernel integrity monitoring GPU program.

Typically, a peripheral device bypasses virtual memory and accesses the system memory directly via physical addressing. To do so, the driver has to create a device-specific mapping of the device's address space that points to the corresponding host's physical pages. In order to create the corresponding OS kernel physical memory mappings to the GPU, a separate loadable kernel module is deployed which is responsible for providing the required page table mapping functionality. As shown in Figure 2, given a kernel virtual address, the loadable kernel module resolves the physical mapping for this address in step 1. In step 2, the kernel module *(i)* allocates one page in the user context and saves its physical mapping, and *(ii)* makes the allocated page point to the same page as the kernel virtual address by duplicating the PTE in the user-page table. Then, in step 3, the kernel module maps this user page to the GPU and gets a device pointer[‡]. Finally, in step 4 the kernel module restores the original physical mapping of the allocation and frees it[§]. By doing so, we are able to effectively map any OS kernel memory page to the GPU address space. Furthermore, the user-allocated page

[‡] `cudaHostRegister()` with the `cudaHostRegisterMapped` flag followed by a call to `cudaHostGetDevicePointer()`.

[§] For memory regions that span multiple pages we need to allocate enough pages and point to them in a sequence, before registering the host-device mapping.

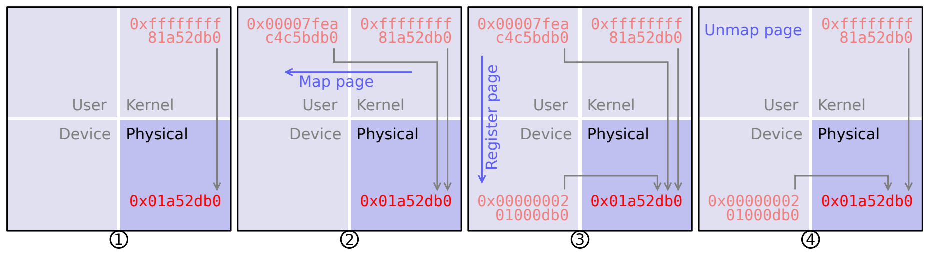

Fig. 2: Mapping OS kernel memory to the GPU. There are several address spaces involved in the operation of GRIM. Initially in step 1 we have a kernel virtual address pointing to a physical address. In step 2 we duplicate this mapping to user space using a kernel module that manipulates page tables. In step 3 we pass the user virtual address to a CUDA API call that pins the page into memory and creates the GPU-side page table entries. In step 4 we destroy the intermediate user space mapping, while the GPU continues to access the physical page.

is unmapped right after the successful execution of the bootstrapping process, in order to destroy all intermediate mappings. We do the same for all kernel virtual memory ranges that we want to monitor with GRIM. The GPU driver populates a device-resident page table for the device to be able to resolve its virtual addresses and perform DMAs.

Modern processor and chipset designs support IOMMUs between peripheral devices and the main memory. Similarly to normal memory management units they translate I/O accesses and provide an extra virtualization layer. Typical uses include contiguous virtual address spaces, extended addressing beyond an I/O device's physical limit, pass-through to virtual machine guests, and memory protection. In GRIM we don't support IOMMUs that perform anything different than 1:1 address re-mapping, at least for the memory address ranges we are interested in, because we don't want our DMA reads from the GPU to CPU-DRAM to go through an IOMMU and get diverted. Furthermore, the IOMMU mappings can be configured by the operating system. For these reasons, we run GRIM with generic CPU-side IOMMUs disabled and all our results are under this assumption.

In order to have unrestricted access to the `/proc/kallsyms` interface, we build Linux with the `CONFIG_KALLSYMS=y` and `CONFIG_KALLSYMS_ALL=y` flags. We note that this is not a requirement of our design, still it helps development and debugging considerably for two reasons: *(i)* it allows us to easily locate the address of the kernel page table instead of explicitly exporting the `init_mm` structure for use by loadable modules, and *(ii)* it saves us the coding of custom memory scanners for all the other data structures we need to locate for monitoring purposes. Obviously, the access to the kernel symbol lookup table might not be acceptable in certain environments. For these cases, we would locate our memory regions using an external symbol table or through memory pattern scanners for dynamic loadable parts, which is certainly feasible.

4.2 Kernel integrity monitoring on the GPU

After the successfully mapping of each requested memory region to the GPU, a daemon GPU program is started. The GPU program hashes all monitored regions and compares the checksums to find changes, in an infinite loop. Due to the non-preemptive execution of the GPU, no other program can execute as long as our endless GPU program is running. As such, it is not feasible to tamper with the GPU on-chip state, such as the provided memory region addresses, the known checksums, and the checksumming code.

The checksumming algorithm can be one of the CRC-32, MD5, or SHA256 (see §5.5 for a comparative evaluation). By default we use the CRC-32 as defined by ISO 3309, due to its simplicity, speed, and its wide adoption. Even though all these algorithms work on byte blocks in an incremental update loop, we have optimized to fetch 16-byte blocks from memory, by using `uint4` typed reads (the widest native data type available). We have also tried to use wider user-defined structs, however it did not improve read performance, because the compiler deconstructs it to operations on the struct's members. To the best of our knowledge, there is no method of issuing larger DMAs from GPU code, using the `cudaHostRegisterMapped` technique.

Instead of maintaining a separate checksum for each memory region, we only keep a single master checksum for all individual checksums. The motivation behind this is to allow the checksum to be stored in registers and remain completely hidden from the host. Even if an attacker is able to stop the execution of the GPU program, the master checksum could not be extracted, due to the fact the GPU registers are automatically reset to zero every time a new program is loaded to the GPU for execution, as has been previously shown [26]. Similarly, the code for CRC32 is small enough to fit in the instruction cache of the GPU, hence an attacker cannot tamper with it [26].

4.3 Real-time notification

Even though the GPU kernel is scheduled to run forever, an adversary that has fully compromised the host, can force the GPU to stop its execution. In order to detect such incidents, we need an external offline host, the admin station, connected directly on the local network via a separate Ethernet port. The admin station monitors the rest of GRIM and is able to power off the host.

The two parties (i.e. the host and the admin station) run a simple protocol based on keep-alive messages. The controlling user process reads a predefined memory area from the GPU, sends the data to the admin station through an established connection, and sets a flag (for the GPU to see) to indicate that the data was sent. The data is a counter together with an alert flag encrypted with a symmetric key. The key is installed to both the GPU and the admin station, at bootstrap, before GRIM starts monitoring the host system. The monitored host has no knowledge of the key and in order to prevent from being leaked, it can be stored either in the GPU registers, which automatically reset to zero every time a new GPU program is loaded [26], or as opcodes in the GPU program's text, which also cannot be retrieved or tampered, as it is completely erased from global memory after bootup and resides only in the non-addressable instruction cache of the GPU. For convenience, we chose to use the former option in our current implementation.

The admin station, assumes that the machine is compromised if a message is missed or if it does not receive a valid message in twice the update period (i.e. 200 milliseconds). Of course a valid message containing an alert also means that the host is compromised. The communication structure that is used between the GPU and the admin station is very simple, as shown in Figure 3. The first two members are sent to the admin station (`data[2]`) and the third (`sent`) is used for communication between the GPU and its controlling program. When the master GPU thread sees that the sent flag is set it increments the counter and encrypts the message data with the key. In case the GPU discovers an illegal checksum, it sets the alert flag and re-encrypts the message. All messages are encrypted using the Extended Tiny Encryption Algorithm (XTEA).

The controlling user process, at the host side, is responsible to read the encrypted message of the integrity monitor and send it to the admin station. This process occurs periodically, every 100 milliseconds, resulting to minimal CPU utilization (lower than 0.1%). If a GPU thread discovers a corrupted region, it indicates it in this message for the admin station to be notified in the next iteration.

```
struct message {
    union {
        int data[2]; /* encrypted data */
        struct {
            int seq;
            int alert;
        };
    };
    int sent; /* plain sent flag */
};
```

Fig. 3: The message format used for synchronization between the GPU, host, and the admin station. When a GPU thread finds a corrupted memory region it sets the alert flag and encrypts it together with the sequence number. A master GPU thread is responsible for incrementing the sequence number when the sent flag is set. The host will send this message to the admin station in the next synchronization point and the alert will be handled there. The host sets the sent flag while the GPU unsets it.

4.4 Data-parallel execution

There are some implementation choices regarding the actual code execution on the GPU. For instance, the partitioning of checksumming work among GPU threads, how synchronization with the host is done, how to do the memory reads.

We chose to have a master thread that, apart from being a normal worker, checks whether the host has sent the packet to the admin station and composes a new message with the incremented sequence number. Furthermore, memory regions are evenly distributed to all threads. During bootstrapping, GRIM finds the greatest common divisor among all region lengths and split larger regions to that size, ending up with equal sized regions. Because of the nature of the problem (checksum computation), we can divide and conquer as we wish. We configure the execution in blocks of 512 concurrent threads, which is the preferred block size. About the memory access pattern, we don't really have room to optimize much because all reads on monitored regions are serviced by the GPU's DMA copy engine(s) and are not local.

5 Evaluation

In this section we evaluate the performance and accuracy of GRIM. We measure the rate for detecting a self-hiding loadable kernel module, as well as the impact that GRIM has on the memory bandwidth of the base system. Furthermore, we show the memory coverage that GRIM can afford, without sacrificing accuracy.

Our experimental platform is based on an Intel Core i7-3770 CPU clocked at 3.4 GHz equipped with 8 GiB of DDR3 DRAM at 1333MHz in a dual-channel configuration. The motherboard is the MSI Z77A-G45. The GPU we use for executing GRIM is an NVIDIA GeForce GTX 770 PCIe 3.0 card with 4 GiB of GDDR5 DRAM. We use a Linux 3.14.23 based system with NVIDIA driver version 343.22 and CUDA 6.5 for the GPU code.

5.1 Self-hiding LKM

Our basic test and case study for determining the accuracy of GRIM is the detection rate of a self-hiding loadable kernel module (LKM). Notice, that this case study is in-line with the evaluation methodology carried out in the state-of-the-art of similar works [18]. Also, the synthetic evaluation we present in this section can *stress* GRIM significantly more than an actual rootkit. The artificial LKM module, which resembles the operation of a rootkit, performs zero malicious operations; it only loads and unloads itself. On the other hand, an actual rootkit, once loaded, needs to perform malicious actions, and therefore it is exposed to the monitor for a longer time period.

Typically, a module is handled by a system utility which is responsible for loading it into memory and invoking a system call to initialize it. Specifically in Linux, the `insmod(8)` and friend tools open the kernel module object file and use the `finit_module(2)` system call with its file descriptor. The system relocates symbols, initializes any provided parameters, adds a handle to data structures (a modules list), and calls the module's `init()` function. A rootkit, implemented as a kernel module, is able to remove itself from that list — in order to increase its stealthiness — and this is typically performed in its `init()` function. Still, this transient change can be detected by carefully monitoring the

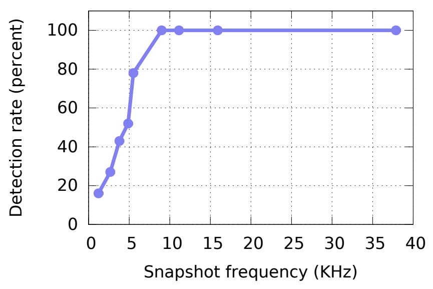

Fig. 4: Self-hiding LKM loading detection with different snapshot frequencies. For each configuration, we loaded a module that deletes itself from the kernel modules list 100 times, while monitoring the head of the list. We achieve 100% detection rate with a snapshot frequency of 9 KHz or more.

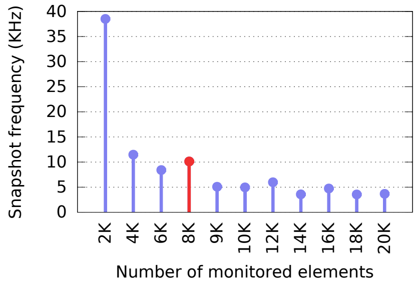

Fig. 5: Maximum achieved frequency depending on the number of pointers being monitored. Increasing the number of (8-byte) memory regions we monitor, lowers the snapshot frequency. Staying above 9 KHz so that we can accurately detect a self-hiding LKM loading lets us monitor *another* 8K pointers.

head of the modules list with a large enough snapshot frequency. In the following experiment, we show the minimal snapshot frequency that is required to perceive such module list changes and thus achieve successful detection.

In order to measure the detection rate, a self-hiding kernel module is loaded, repeatedly, 100 times, using a different snapshot frequency. Figure 4 shows the detection rate achieved by GRIM under each configuration. We can see that GRIM can reliably detect that a new kernel module has been loaded before hiding itself with a snapshot frequency of 9 KHz or more, achieving 100% detection rate. That means that GRIM detected all module insertions and generated exactly 100 alerts. Note that according to the state-of-the-art, a snapshot-based approach can deliver a 70% detection rate at best [18].

The experiment we carry out in this paper for demonstrating the high levels of detection rate that can be achieved using GRIM is designed in analogy with the one presented in the evaluation of KI-Mon [18]. We have just omitted the verification part of the observed change. KI-Mon, once a change is detected, further verifies semantically if the change is meant to be malicious or not. This verification procedure happens *using snapshots* and in parallel with the detection algorithm, which is based on snooping. We argue that detection and verification are orthogonal. GRIM is fully programmable and can be easily extended with rich functionality for applying *in-parallel* semantic verification algorithms once a change is detected *without* decreasing its detection rate.

5.2 Address space coverage

Next we study the implications of requiring a snapshot frequency of at least 9 KHz for accurate detection, with respect to the amount of memory we can cover. The snapshot frequency is a function of the number and size of the monitored memory regions. Also, alignment plays a role in the read performance of the GPU, 16-byte aligned reads being the fastest. We don't, however, control the placement of the kernel data structures, and thus we assume that some of our monitored regions need one or two extra fetches. Given our specific implementation, the most efficient read width is 16 bytes (or one `uint4` in CUDA's device native data types). In the following experiment we focus on monitoring pointer values (8-byte regions). The results are shown in Figure 5. Given our detection rate results, we see that we can monitor at most 8K pointers simultaneously without sacrificing accuracy, because we need to stay above the 9 KHz snapshot frequency. This limits the address space we can monitor using GRIM if we want to achieve 100% detection rate, albeit 8K addresses spread out in memory could potentially safeguard many important kernel data structures. Moreover, this is not an inherent limitation that only GRIM suffers from. All hardware-based integrity monitors [13, 18, 22] can observe only a limited fraction of the host's memory. Even snoop-based systems need to filter out most of the memory traffic which targets memory that is not a potential target for a rootkit.

5.3 Impact on memory bandwidth

In this section we measure the overhead that GRIM adds to the memory subsystem. To do so, we ran the STREAM benchmark [20] while the system is under monitoring by GRIM and when the system is idle. We use all 8 CPU threads for the benchmark and we run our GPU snapshot loop with different throttling factors to obtain various frequency values. We count the total number of memory references by multiplying with the obtained snapshot frequency. We show the results in Figure 6. At most 17% of the available memory bandwidth is utilized by the GPU when GRIM is in place. Note, that the system consumes 17% of the available memory bandwidth in the worst-case, in which GRIM is monitoring 8K of 8-byte memory elements. As we show in Figure 7, monitoring of 512 8-byte memory elements (enough for safeguarding the system-call table) consume only 1% of the memory bandwidth. For this particular experiment we throttled our snapshot loop to approximately get to the desired snapshot frequency of 9 KHz for different number of monitored regions (again of 8 bytes in size). We note that we can always limit the host memory bandwidth degradation by monitoring less pointers. Therefore, we stress that *(i)* our system is flexible and can adapt its configuration for consuming less memory bandwidth and safeguarding less memory if this is desirable, and *(ii)* even in the worst case, when GRIM is monitoring 8K 8-byte elements, it consumes 17% of memory which is comparable with the memory consumption reported by similar systems [13].

5.4 Using a low-end GPU

Given that the snapshot process is I/O bound on the DMA path, we also explore the behaviour of GRIM on low-end GPUs. To do so, we use a NVIDIA GT 630 PCIe 2.0 and measure the memory coverage that can afford while maintaining a

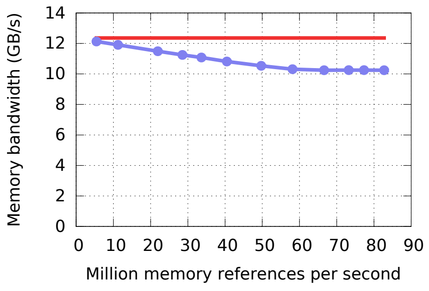

Fig. 6: Impact on memory bandwidth while the system is under monitoring. The GPU issues DMAs which contend with the CPU cores on the memory controller and/or the DRAM itself, limiting the memory bandwidth normally available to host. GRIM degrades STREAM performance by 17% in the worst case.

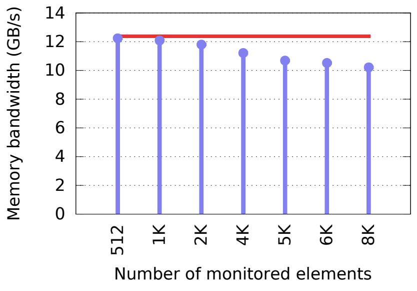

Fig. 7: Available memory bandwidth with respect to memory coverage. Snapshotting is throttled to approximately achieve the required snapshot frequency of 9 KHz. We see that monitoring 512 8-byte elements only consumes 1% of the memory bandwidth whereas with 8K we reach the 17% worst case.

detection rate of 100%. The GT 630 is able to reliably monitor at most 2K 8-byte elements (without sacrificing detection accuracy). Even though this is 4 times lower than the detection rate sustained by the GTX 770, it comes with great benefits in terms of energy efficiency. Figure 8 shows the power consumption of each device while being idle and the active power consumption while executing the GPU component of GRIM. The low-end GPU draws almost 6 times less power both when running our code and in total when taking into account the idle power consumption. That creates an interesting trade-off and makes the low-end choice attractive for setups with low power budgets. Finally, the GT 630 can only affect STREAM performance by 6% in the worst case due to its host connectivity limitations.

	idle	active
GTX 770	67.84	148.87
GT 630	11.85	25.70

Fig. 8: Power consumption of each device in Watts while being idle, as well as including the additional power the device draws while GRIM is running ("active" column). We observe that the low-end GPU consumes almost 6 times less power both when idle and active.

5.5 Checksums and message digests

In the results we showed so far we have been using CRC32 to detect memory modifications. CRC32 has low complexity in terms of computation, while it is

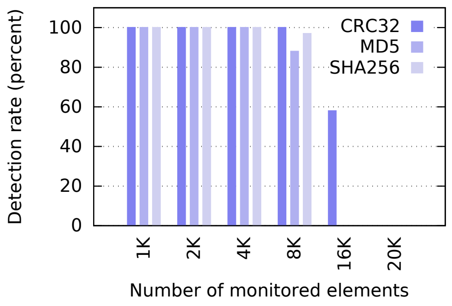

Fig. 9: Comparison of the LKM-hiding detection rate of the CRC32 checksum and the MD5/SHA256 digests for different number of monitored 8-byte elements. The CRC32 is faster and can cover a larger amount of memory without sacrificing accuracy. MD5/SHA256 on the other hand provide higher security.

not considered cryptographically secure. Here we show the overheads involved in using MD5 or SHA256, and the impact it has to detection rate. We monitor different counts of 8-byte elements and show how accurately we can detect the LKM-hiding attack when one of the elements is the head pointer of the modules list. Figure 9 shows that MD5 performs a little worse than CRC32 sustaining 4K elements but losing accuracy at 8K elements. The same is true for SHA256. We expect MD5 to be faster than SHA256 but here we test with relatively small data blocks so read performance is more critical than actual computation. Both implementations of MD5 and SHA256 cannot detect the LKM memory-write when the monitor inspects 20K elements. This is a trade-off between memory coverage and collision resistance, which can be configured. Notice that for GRIM even a simple algorithm, like CRC32, can be quite effective in detecting kernel rootkits, unless the malicious code can perform operations on memory by preserving the CRC32 checksum of the particular modified memory page, which is not trivial.

6 Related work

Integrity monitors have formed an attractive technology for protecting software against exploitation. Based on monitoring, they can *infer* about an attack and not *defend* against the attack. As an alternative method of protection, integrity monitors are considered promising, especially when even advanced defenses, such as preventing code-reuse in the operating system [19] can be bypassed by advanced attacks [15,16], and when it has been demonstrated that core protection principles, like Control-Flow Integrity (CFI) applied at the kernel [8], offer limited security [9].

Integrity monitors can be implemented in both software and hardware. Software integrity monitors [6, 10, 25, 29] are based on hypervisors. The operating system runs as a guest and is occasionally checked by the hypervisor for possible (malicious) modifications. Although these monitors dramatically limit the code base that should be trusted and bug-free, there is always the possibility for the hypervisor to be exploited. The hypervisor and the operating system are not completely isolated and they are both written in untrusted languages. Of course, all these solutions are towards the right direction, and it is obviously easier to perform a security analysis in a significant smaller program (the monitor) compared to protecting the complete operating system. However, the community has been in parallel seeking for more solid monitors, which will be physically isolated from the rest of the system, they will be implemented using custom hardware, and won't run in the same code base with the kernel they protect.

As we have stressed throughout the paper, we offer an integrity monitor based on GPUs, which closely matches the work demonstrated by hardware monitors such as Copilot [13], Vigilare [22], and KI-Mon [18]. Copilot [13] is a snapshot-based monitor implemented on a custom FPGA. Essentially, GRIM offers all of the Copilot's functionality, in addition to better performance and extensibility, since GRIM is fully programmable. Vigilare [22], on the other hand, argues that it is hard to achieve good detection rates using snapshot-based monitors and thus it introduces snooping, i.e., monitoring the bus for particular memory operations that can affect the kernel structure. We believe that snooping is important, and certainly a lightweight check compared to the snapshot-based approach, however, in this paper, we argue that snapshot-based monitors can do significantly better. With GRIM we are able to achieve 100% detection rate. Finally, KI-Mon [18] extends Vigilare by offering protection for mutable objects. In GRIM we can protect against mutable objects, however we have not implemented the *semantic verification check* for validating if a change of a mutable object in the kernel is the result of a legitimate operation or not. We omitted implementing this in GRIM, because the available API for programming the GPU is proprietary and limited. Nevertheless, as we have in detail discussed, our architecture can support this operation.

7 Conclusion

In this paper we revisited snapshot-based Kernel Integrity Monitors (KIMs) and we demonstrated that a novel GPU-based architecture can do substantially better than it has so far been reported in the state-of-the-art. GRIM builds on commodity GPUs and offers a fully extensible and programmable architecture for implementing complex KIMs. Our thorough evaluation of GRIM suggests that we can achieve 100% detection rate of evolved rootkits that try to evade snapshot-based monitors. This detection rate outperforms the currently reported rate (70%) of the state-of-the-art of hardware-based monitors.

GRIM offers an attractive option for instantly deploying a hardware-based KIM. It needs no modifications to the host it protects, no kernel recompilation or installation of custom hardware. This is particular important, because all so far proposed hardware monitors that base their operation on snooping require changes at the microprocessor level. In our case, GRIM acts as a secure co-processor that protects a vulnerable host from malicious rootkits. We believe our proposal will further promote research in the field of advanced KIMs that are snapshot-based, since there is clearly enough space for optimizations and many benefits to be considered when it comes to deployment.

8 Acknowledgements

We thank our shepherd Zhiqiang Lin and the anonymous reviewers for their invaluable feedback. This work was supported by European Commission through the H2020 ICT-32-2014 project SHARCS under Grant Agreement number 644571.

References

1. Nouveau driver for nVidia cards. `http://nouveau.freedesktop.org/`.
2. NVIDIA Developer Forums - CUDA kernel timeout. `https://devtalk.nvidia.com/default/topic/417276/cuda-kernel-timeout/`.
3. OpenCL. `http://www.khronos.org/opencl/`.
4. PathScale NVIDIA graphics driver. `https://github.com/pathscale/pscnv`.
5. shinpei0208 / gdev. `https://github.com/shinpei0208/gdev`.
6. A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *CCS*, 2014.
7. S. Chen, J. Xu, and E. C. Sezer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security*, 2005.
8. J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Security and Privacy*, 2014.
9. E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. Security and Privacy, 2014.
10. O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, 2011.
11. R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security*, 2009.
12. D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. ATRA: Address Translation Redirection Attack against Hardware-based External Monitors. In *CCS*, 2014.
13. N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security*, 2004.
14. S. Kato. Implementing Open-Source CUDA Runtime. 2013.
15. V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. Ret2Dir: Rethinking Kernel Isolation. In *USENIX Security*, 2014.
16. V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *USENIX Security*, 2012.
17. G. Klein, P. Derrin, and K. Elphinstone. Experience report: sel4: formally verifying a high-performance microkernel. In *ACM Sigplan Notices*, volume 44, pages 91–96. ACM, 2009.

18. H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security*, 2013.
19. J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys*, 2010.
20. McCalpin, John. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `https://www.cs.virginia.edu/stream/`.
21. K. Menychtas, K. Shen, and M. L. Scott. Enabling OS Research by Inferring Interactions in the Black-box GPU Stack. In *USENIX ATC*, 2013.
22. H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *CCS*, 2012.
23. NVIDIA. CUDA Programming Guide, version 4.0. `http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf`.
24. J. Rutkowska and A. Tereshkin. Bluepilling the xen hypervisor. *Black Hat USA*, 2008.
25. A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, 2007.
26. G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *CCS*, 2014.
27. G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *CCS*, 2011.
28. S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *USENIX Security*, 2014.
29. J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *RAID*, 2010.