We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS
BOOK CITATION INDEX
INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Security Applications of GPUs

Giorgos Vasiliadis

Abstract

Despite the recent advances in software security hardening techniques, vulnerabilities can always be exploited if the attackers are really determined. Regardless the protection enabled, successful exploitation can always be achieved, even though admittedly, today, it is much harder than it was in the past. Since securing software is still under ongoing research, the community investigates detection methods in order to protect software. Three of the most promising such methods are monitoring the (i) network, (ii) the filesystem, and (iii) the host memory, for possible exploitation. Whenever a malicious operation is detected then the monitor should be able to terminate it and/or alert the administrator. In this chapter, we explore how to utilize the highly parallel capabilities of modern commodity graphics processing units (GPUs) in order to improve the performance of different security tools operating at the network, storage, and memory level, and how they can offload the CPU whenever possible. Our results show that modern GPUs can be very efficient and highly effective at accelerating the pattern matching operations of network intrusion detection systems and antivirus tools, as well as for monitoring the integrity of the base computing systems.

Keywords: security, network security, host security, intrusion detection, antivirus, kernel integrity monitoring

1. Introduction

The ever-increasing amount of malicious software (malware) constitutes an enormous challenge to network operators, IT administrators, as well as ordinary home users. To protect against such an evolving threat landscape, it is necessary to provide detection of malicious activities at different levels: (i) by inspected exchanged data at central network traffic ingress points, (ii) by scanning of unwanted software at the storage level, and (iii) by providing program memory integrity at the host level. Three of the most widely used tools that perform such kind of operations are intrusion detection systems, antivirus software, and host integrity tools. Unfortunately, the constant increase in link speeds, storage capacity, number of end-devices and the sheer number of malware, poses significant challenges to all these tools, which end up requiring high scanning throughput and low latency.

Typically, the detection of malicious activities spends the majority of its time matching data streams against a large set of known signatures or checksums, using string searching, regular expression matching and hashing algorithms. Signature matching algorithms analyze the data stream and compare it against a database of fixed strings or regular expressions to detect known malware. The signature patterns can be quite complex, composed of wild-card characters,

range constraints, different-size strings, and sometimes recursive forms. To make matters worse, the number of signatures is increasing proportional every year, as the amount of malware grows, exposing scaling problems of anti-malware products.

Modern GPUs have been proven to be highly effective and very efficient at accelerating computational- and memory-intensive workloads. The ever-growing video game industry is a driving factor for becoming ever more powerful and flexible stream processors, specialized for highly parallel operations. Comparing with commodity CPUs, the massive number of transistors is devoted to data processing, rather than data caching and flow control, making them ideal to perform data parallel computations that up till now were handled by the CPU.

In this chapter, we present new models for malware detection tools that operate at the network, storage, and memory level. These models combine the commodity, general-purpose GPU paradigms, tailored for high-performance and low-latency analysis. Our systems take advantage of the parallelism offered by the GPUs to improve scalability and runtime performance and are able to offload the CPU whenever possible. Our results show that modern GPUs can be highly effective and very efficient at accelerating a highly diverse set of operations that are core functions of modern security tools, including string searching, regular expression matching and checksum computations.
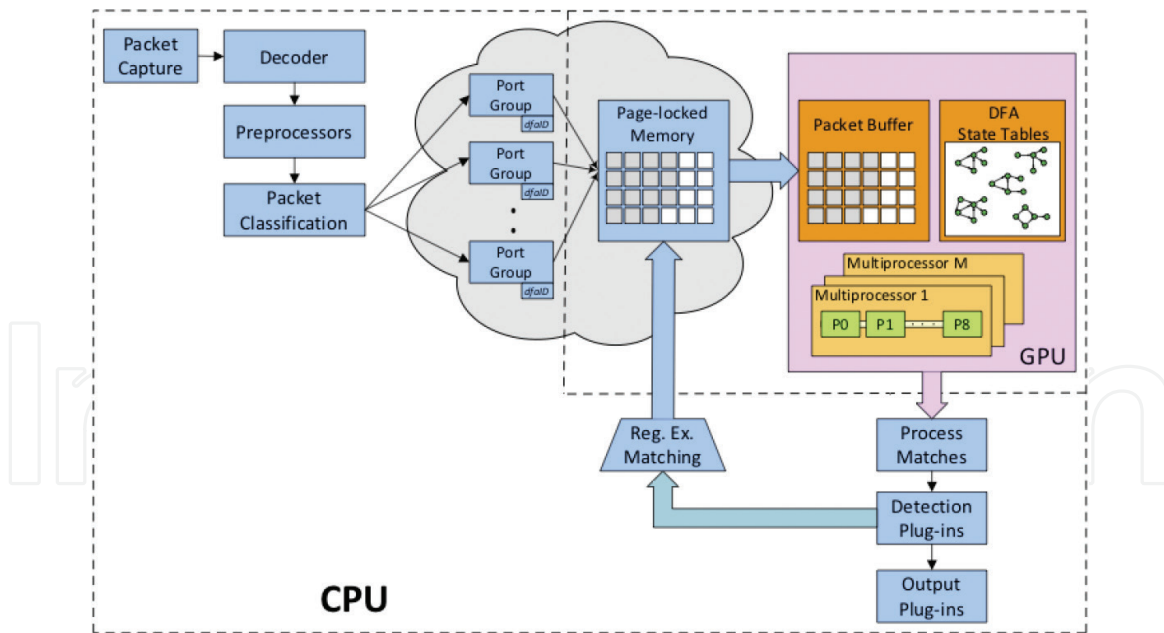
2. Network intrusion detection and prevention systems

First, we show how to exploit the parallelism of the graphics processing unit (GPU) to offload specific intensive tasks of a network intrusion detection system (NIDS). Particularly, we present the design, implementation, and evaluation of string searching and regular expression algorithms engines running on GPUs. We have integrated these implementations in the popular Snort intrusion detection system [1] to offload both string and regular expression matching computation, as shown in **Figure 1**.

The data parallel capabilities of modern GPUs can allow the concurrent matching of multiple input data streams at the same time against a large set of fixed string patterns and regular expressions. Mainly, the architecture can be separated in several different tasks: packet capturing, decoding, preprocesing, the transfer of the network packets to the GPU, the string-matching on the GPU, and the transfer of the matching results back to the CPU, where all the remaining conditions of the detection rules are checked. Whenever a packet needs to be scanned against a regular expression, it is subsequently transferred back to the GPU where the actual matching takes place.

2.1 Architecture

The overall design of our GPU-assisted network intrusion detection architecture, has two key factors for achieving good performance: (i) load balancing between processing units, and (ii) linear performance scalability with the addition of more processing comuts. In particular, the monitored traffic is distributed at the flow-level to different CPU cores, by applying a symmetric hash function on the 5-tuple fields of each packet header (i.e., source IP address, destination IP address, source port, destination port, protocol). Eventually, all packets of the same flow (i.e., same connection) will always be placed in the same ring buffer, and will be processed by the same CPU-core. This inherently leads us to a multi-core architecture, in which each core processes an evenly distributed portion of
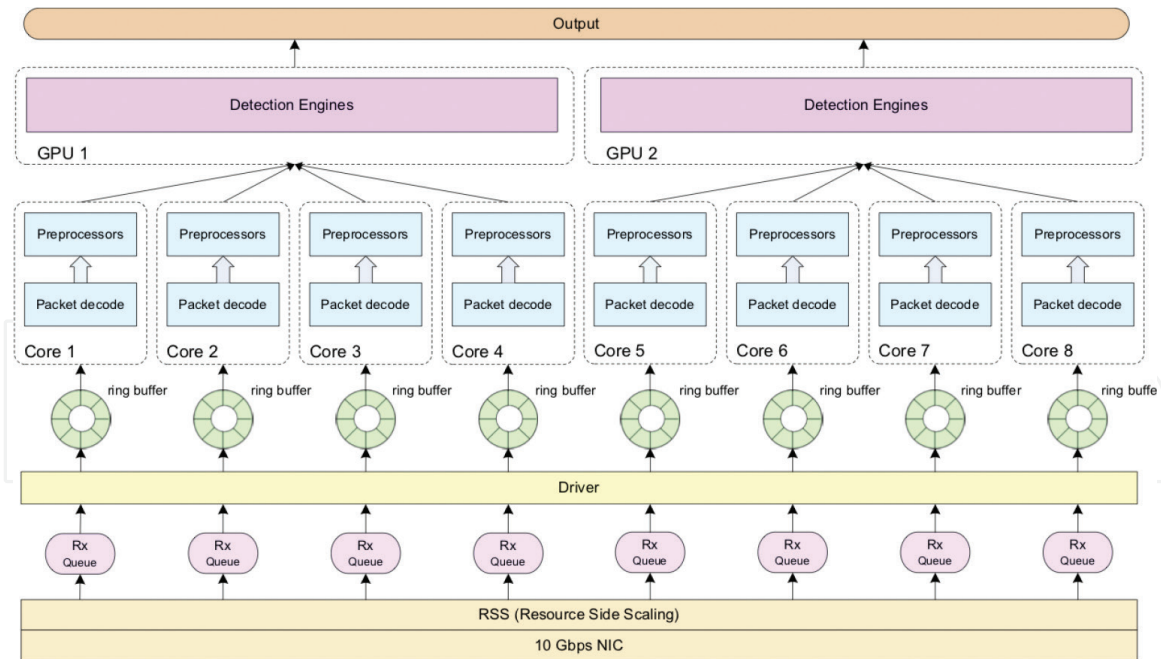
Figure 1.
Overview of the single-threaded GPU-based network intrusion detection architecture.

the network traffic, without requiring any intra-node communication for processing operations that are limited in scope to a single flow. Each CPU core is responsible for network flow tracking, protocol parsing, TCP stream reassembly, and content normalization. The reassembled and normalized packets of each network flow are then transferred to the graphics card in large batches, in order to be processed in parallel. This enables an "intra-flow" parallelism, in which network packets from the same flow can be processed in parallel, while also maintaining flow-state dependencies. We note that this buffering scheme, as well as the extra data transfer operations that need to be performed between the memory address spaces of each device obviously, adds some latency to the processing path. Even though the computational gains offered by the GPU tolerates these extra data transfers and pay off in terms of increased throughput, we further mitigate these overheads by implementing pipelining schemes that allow the CPU and GPU execution to overlap, thus offering an additional level of parallelism to the overall execution path (see Section 2.1.3). Overall, by parallelizing both packet preprocessing and content inspection across multiple CPUs and GPUs, our proposed architecture can operate in multi-Gigabit networks using solely commodity components.

As shown in **Figure 2**, we utilize the different processing units available (i.e., CPUs and GPUs) in order to map the different functionalities that are performed across the incoming network flows, using both *task* and *data* parallelism. More specifically, the network interface distributes the incoming network packets to the CPU-cores, by applying a symmetric hash function on the 5-tuple fields of each packet header (i.e., source IP address, destination IP address, source port, destination port, protocol). This ensures that all packets of the same flow (i.e., same connection) will always be placed in the same ring buffer, and will be processed by the same CPU-core. Each CPU-core reassembles and normalizes the captured traffic before offloading it to the GPU for pattern matching [2]. Any matching results are logged by the corresponding CPU-core using the specified logging mechanism, such as a file or database.

This design has many advantages: *First*, no synchronization or lock mechanisms is needed, since different network flows will be processed by different CPU-cores independently. *Second*, each CPU-core maintains smaller data

Figure 2.
The architecture of the GPU-enabled network intrusion detection system.

structures (e.g., the flow management table, the TCP reassembly tables, etc.) instead of sharing a few large ones, which reduces both the number of tables lookups, as well as the size of the working set in each cache, increasing overall cache efficiency.

2.1.1 Parallel multi-pattern engine

A major design criterion for scanning large network data flows against many different fixed string patterns, is the choice of an efficient matching algorithm. The majority of network intrusion detection systems utilize a flavor of the Aho-Corasick algorithm [3] for string searching. Internally, the algorithm uses a transition function that computes the next state T [state, c] for a given state and a character c. A pattern is matched every time the algorithm transits to a final state. The performance and memory requirements of Aho-Corasick depend on how the transition function is implemented. In the full implementation, hereinafter AC-Full, each transition is represented with 256 elements, one for each 8-bit character. Each element contains the next state to move to, as a result the next state can be found in $O(1)$ steps for every input character; this ensures a linear complexity over the input data, independently on the number of patterns, which is very efficient in terms of performance.

However, a disadvantage of the full state representation is the large memory requirements, even for small signature sets. For Snort, the compiled state table can reach up to several hundred Megabytes of memory. To make matters worse, CPU processes cannot share memory on the GPU device, as such a different memory space has to be allocated in the GPU. This can result to significant memory allocations, as shown in **Table 1**. Given that the GeForce GTX780 that we used for our evaluation comes with 2GB of memory, only two Snort instances can fully utilize the GPU at a time.

To preserve scalability with respect to the number of concurrently running Snort instances, it is important to optimize the memory requirements needed. As such, instead of using the full state table representation, we use a compacted version, similar to [4], in which the states are represented in a banded-row format. In particular, we store only the elements from the first non-zero value to the last

#Rules	#Patterns	#States	AC-Full	AC-Compact
8192	193,167	1,703,023	890.46 MB	24.18 MB

Table 1.
Memory requirements of AC-Full and AC-Compact for the default Snort rule set.

non-zero value of the table; the number of the stored elements is known as the bandwidth of the sparse table. Obviously, in the compacted implementation, namely AC-Compact, the next state cannot be accessed directly while matching input bytes. Instead, it has to be computed, as shown in **Figure 3**; this computation obviously adds a small overhead at the scanning phase, which is amortized though by the significantly lower memory consumption.

Moreover, it is common that some patterns may share the same final state in the state table or be case-insensitive. Instead of adding every different combination of capital and lower letters for every case-insensitive pattern, we simply mark that pattern as case-insensitive and add only one combination (i.e., all characters are converted to capitals). In case the pattern is matched in a packet, an extra case-insensitive search should be made at the index where the pattern was found. Similarly, if two patterns share the same final state, they need to be verified in case of a match.

Each GPU thread processes a different reassembled network packet. We use an array to store the network packets; every time the array fills up, it is transferred to the GPU and processed at once. **Figure 4** shows the sustained throughput when matching full-size packets (i.e., of 1500-bytes length) on a single GTX770, for a varied number of packets that are processed at once. The traffic is generated from a separate machine over four 10 Gbit/s network cards. As can be shown, the AC-Full achieves a peak performance of 21.1 Gbit/s, while the AC-Compact about 16.4 Gbit/s. In both cases, all data transferring costs to and from the GPU are included. The corresponding CPU implementation achieves a performance of 0.6 Gbit/s for the AC-Full implementation, and thus a single GPU instance corresponds to 36.2 and 28.1 CPU-cores for the AC-Full and AC-Compact implementations, respectively.

As expected, AC-Full outperforms AC-Compact in all cases. The added overhead of the extra computation that AC-Compact performs in every transition decreases its performance about 30%. The main advantage of AC-Compact is that it has significantly lower memory consumption than AC-Full. The corresponding memory requirements for storing the detection engines of a single Snort instance are shown in **Table 1**. As shown, AC-Compact utilizes up to 36 times less memory, which makes it a better fit for a multi-CPU environment, due to CUDA's limitation of allocating a separate memory context for each host thread. Using AC-Compact, a single GTX770 card can store the detection engines of about 80 Snort instances (80 × 24.18 MB ≈ 1.9 GB). The remaining memory can be used for storing the actual contents of the incoming network packets. If AC-Full is used, only two instances can fit in device memory.

2.1.2 Compiling PCRE regular expressions to DFA state tables

The majority of tools that use regular expressions typically convert them into DFAs [5]. To do that, the most common approach is to first compile them into NFAs, and then convert them into DFAs. We also follow the same approach, and, using the Thompson algorithm [6], we first convert each regular expression into an NFA. The generated NFA is then converted incrementally to an equivalent DFA, using the Subset
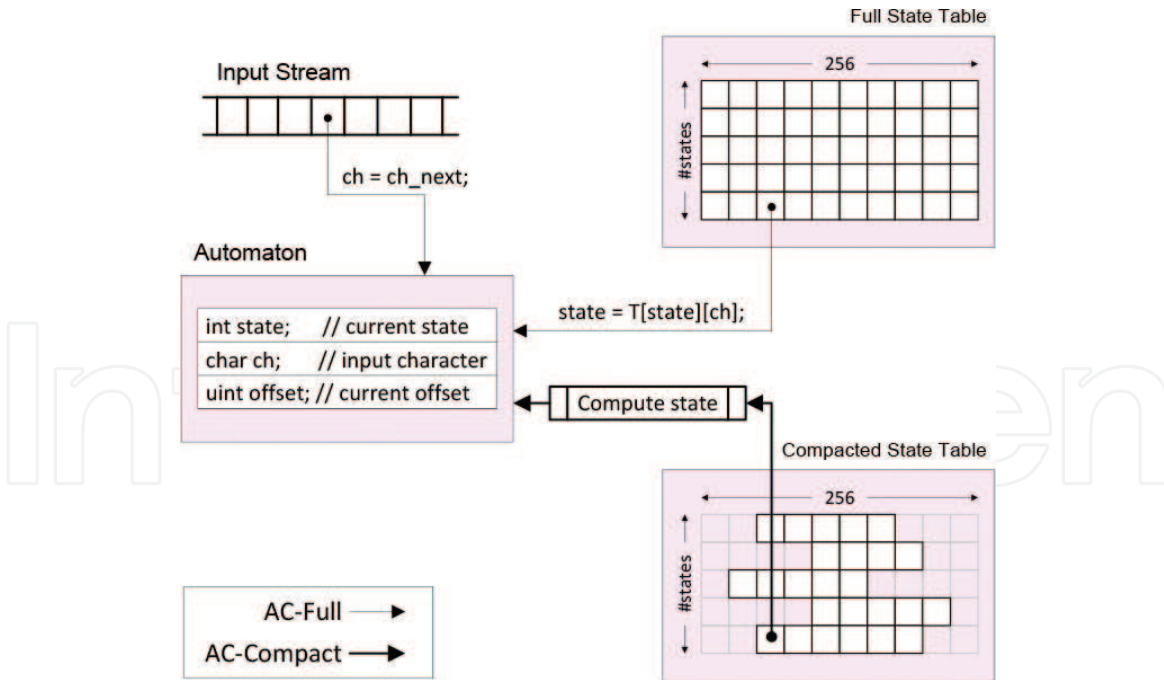
Figure 3.
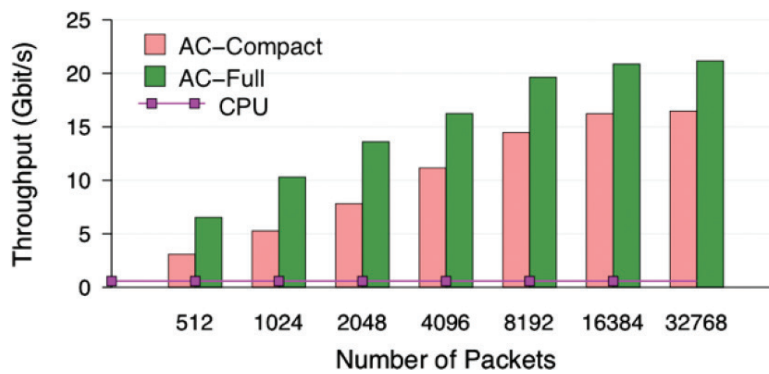*State tables of AC-Full vs. AC-Compact.*

Figure 4.
GPU throughput for AC-Full and AC-Compact.

Construction algorithm. The basic concept of subset construction is to define a DFA in which each state is a set of states of the corresponding NFA. Each state in the DFA represents a set of active states in which the corresponding NFA can be in after some transition. During the matching phase, the resulting DFA achieves $O(1)$ computational cost for each incoming character.

However, a major concern when converting regular expressions into DFAs is the state-space explosion that may occur during compilation [7]. To distinguish among the states, a different DFA state may be required for all possible NFA states. Obviously, this can result to exponential growth of memory utilization, primarily due to the usage of wildcards, e.g. (.*), and repetition expressions, e.g. (a (x,y)). As a result, certain regular expressions may end-up consuming large amounts of memory when compiled to DFAs. A theoretical worst-case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(\Sigma^n)$ states, where Σ is the size of the alphabet, i.e. 2^8 symbols for the extended ASCII character set [8].

To prevent the greedy memory consumption that can be occurred by some regular expressions, we follow a hybrid approach, in which we convert only the regular expressions that do not exceed a certain threshold of states; the remaining regular expressions will be matched on the CPU using NFAs. The total number of states is traced during the incremental conversion from the NFA to

the DFA and the conversion stops if a certain threshold is reached. As we have experimentally found, more than 97% of the total regular expressions used by Snort can be converted to DFAs, when using an upper bound of 5000 states per expression. The remaining expressions can be processed by the CPU using an NFA implementation, similar to the vanilla Snort.

Each DFA is represented as a two-dimensional array that is mapped linearly on the memory space of the GPU. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively. Each cell contains the next state to move to, as well as an indication of whether the state is a final state or not. The final states are represented as negative numbers, due to the fact that transition numbers can be positive integers only. Whenever the state machine reaches into a state that is represented by a negative number, it considers it as a final state and reports a match at the current input offset.

2.1.3 Multi-GPU support

Our system is able to utilize several GPUs simultaneously, by dividing the incoming flows equally and performing the signature matching concurrently across all devices. In the CUDA runtime system, each device is bound to a single process. As such, several host processes must be spawned (at least one process per device) in order to enable multi-GPU support. The load balancing scheme shown in **Figure 2** ensures that each process receives a uniform amount of flows; as a result, flows are equally distributed to the different GPUs. By default, our system utilizes all the GPUs that are available in the system, still this can be easily configured by defining the number of GPUs it should try to use.

3. Host-based virus scanning

Antivirus software is one of the most popular tools for detecting and stopping malicious or unwanted software. ClamAV [9] is a popular open-source virus scanner, which contains more than 60 thousand signatures, formed by both fixed strings, as well as regular expressions. It can be used both at the server-side for protecting mail and file servers, as well as for client personal computers. The database includes signatures for polymorphic viruses in regular expression format and for non-polymorphic viruses in simple string format. To detect non-polymorphic viruses, the current version of ClamAV uses an optimized version of the Boyer-Moore algorithm [10] for matching simple fixed string signatures. For polymorphic viruses, ClamAV uses a variant of the classical Aho-Corasick algorithm [3].

The main design principle of our GPU-assisted antivirus is to utilize the GPU in order to quickly filter out the data segments that do not contain any viruses. To achieve this, we have modified ClamAV, such that the input data stream is initially scanned by the GPU. The GPU uses a prefix of each virus signature to quickly filter-out clean data. The motivation behind this is that the majority of the data do not contain any viruses, as such the GPU filtering is quite efficient, as shown in **Figure 5**.

Figure 6 presents the overall architecture of our GPU-assisted antivirus tool. The contents of each file are read from disk and stored into a file buffer. The file buffer is used to store the contents of many files, and is transferred to the GPU in a single transaction. This results in a reduction of I/O transactions over the PCI Express bus. Moreover, the file buffer is page-locked (i.e., it does not get swapped), hence it can be transferred asynchronously, via DMA (Direct Memory Access), to the memory space of the GPU.
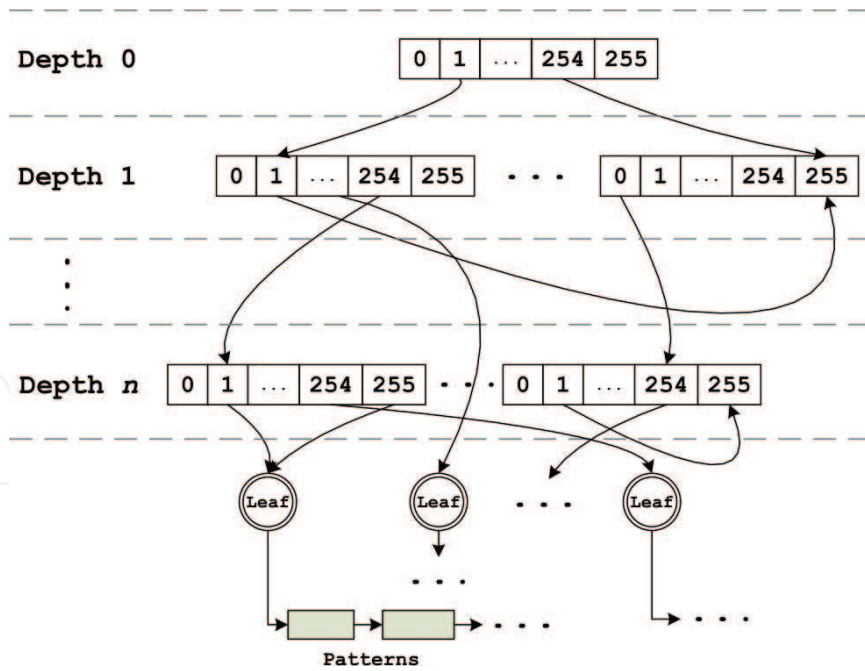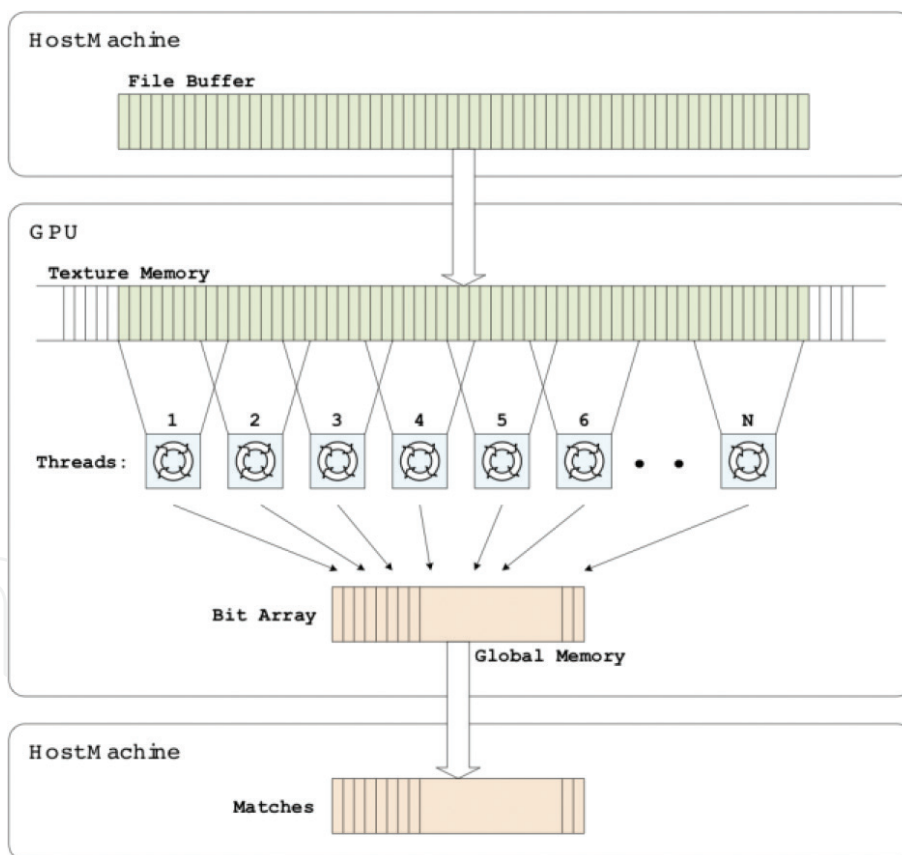
Figure 5.
Number of matches.

Figure 6.
The architecture of the GPU-assisted antivirus. Files are mapped onto pinned memory that can be copied onto the graphics card via DMA. A first-pass filtering is performed on the GPU and return any potential true positives for further checking onto the CPU.

Every time the GPU detects a suspicious virus, i.e., there is a prefix match, the file is further investigated by the verification module. Otherwise, no further computation takes place. The data parallel operation of the GPU is ideal for creating multiple search engine instances that will scan for virus signatures on different

data in a massively parallel fashion. Overall, the GPU is employed as a high-speed first-pass filter, before completing any further potential signature-matching work on the CPU.

3.1 Basic mechanisms

Initially, the entire signature set of ClamAV is preprocessed, in order to construct a deterministic finite automaton (DFA). As we have explained before, the DFA state machine provides linear complexity over the input text stream, which is very efficient. Unfortunately, it is not feasible to construct the full DFA, due to the big number and large size of the virus signatures contained in the ClamAV.

To overcome this, we use a portion from each virus signature for constructing the DFA, and specifically only the first n symbols, similar to [11]. By doing so, the height of the resulting DFA machine is limited, as shown in **Figure** 7. Any patterns that share the same prefix are stored under the same final node, called leaf. In case the length of the signature pattern is smaller than the prefix length, the entire pattern is added. A prefix may also contain special characters, such as the wild-characters * and ?, that are used in ClamAV signatures to describe a known virus.

At the scanning phase, the input file data will be first scanned by the DFA running on the GPU. It is clear that the DFA may not be able to match an exact virus signature inside a data stream, as in most cases the length of the signature is longer than the length of the prefix we used to create the automaton. This will be the first-level filtering though, which purpose is to use the high parallelism of the GPU to quickly filter-out the majority number of true negatives, and drastically eliminate a significant portion of the input data that need to be scanned by the CPU. Obviously, the longer the prefix, the fewer the number of false positives at this initial scanning phase. As shown in **Figure** 5, using a value of 8 for n, can result to less than 0.0001% of false positives in a realistic corpus of binary files.

3.2 Parallelizing DFA matching on the GPU

During scan time, the algorithm iterates over the input data stream one byte at a time and moves the current state appropriately. The pattern matching is performed byte-wise, meaning that we have an input width of 8 bits and an alphabet size of $2^8 = 256$. Thus, each state will contain 256 transitions to other states, as shown in **Figure** 7. If the scanning algorithm reaches a final-state, then a potential signature
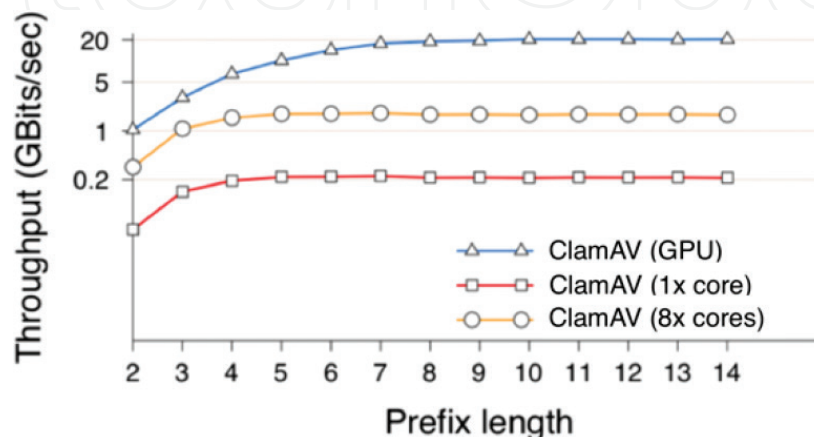
Figure 7.
A fragment of the DFA structure with n levels. All the patterns that begin with the same prefix are listed under the same leaf (final node).

match has been found, and the corresponding offset is marked. All marked offsets will be verified later by the CPU.

To utilize all streaming processors of the GPU, we exploit its data parallel capabilities by creating multiple threads. An important design decision is the partitioning of the input data to each thread. The simplest approach would be to use multiple data input streams, one for each thread, in separate memory areas. However, this will result in asymmetrical processing effort for each processor and will not scale well. For example, if the sizes of the input streams vary, the amount of work per thread will not be the same. This means that threads will have to wait, until all have finished searching the data stream that was assigned to them. To overcome this, we use a single input data stream and each thread searches a different portion of it. In particular, our strategy splits the input stream in distinct chunks, and each chunk is processed by a different thread. **Figure 8** shows how each GPU thread scans its assigned chunk, using the underlying DFA state table. Although they access the same automaton, each thread maintains its own state, eliminating any need for communication between them.

The two operations of DFA matching, is determining the address of the next state in the state table and fetching the next state from the device memory. These memory transfers can take up to several hundreds of nanoseconds, depending on the memory congestion. To obtain the highest level of performance and hide memory latencies, we run many threads in parallel. Multiple threads can overlap data transfer with computation, hence improving the memory bandwidth.

Moreover, we have explored storing the DFA state table both in the global memory space, as well as in the texture memory space of the GPU. The texture memory can be accessed in a random fashion for reading, in contrast to global memory, where the access patterns must be coalesced. This feature can be very useful for algorithms like DFA matching, which exhibit irregular access patterns across large data sets. As described in Section 2.1.2, the usage of texture memory can boost the computational throughput up to a factor of two.

A case that requires special consideration though, is when patterns span across two or more different chunks. The simplest approach for fixed string patterns, is to continue the scanning to the next chunk (s), up to n bytes, where n is the maximum pattern length in the dictionary. However, the patterns used for virus scanning are typically very large, especially compared with other signature-based tools, such as Snort. Besides that, a virus signature may contain wild card characters (i.e., *), as such the length of the patterns may not be determined. To overcome this, the following heuristic is used: each thread carries on the search to the consecutive bytes of the following chunk, up till a fail or final-state is reached. While matching a pattern that spans chunk boundaries, the state machine will perform regular transitions. However, if the state machine reaches a fail or final-state, then it is clear that there is no need to continue the searching, since any consecutive patterns will be found by the thread that was assigned to search the current chunk. This enables us to avoid any communication between the threads concerning boundaries in the input data buffer. Every time a match is found, it is stored to a bit array. The size of the bit array is equal to the size of the data that is processed at once, and each bit represents whether a match was found in the corresponding offset.

Figure 9 shows the throughput achieved for different prefix lengths. As input data stream, we use the files under /usr/bin/ of a typical Linux installation, which contains 1516 binary files of about 132 MB. To eliminate disk latencies, all files are cached in memory by the operating system. Even though the files do not contain any viruses, they exercise most code branches of our tool. As we can see, the overall
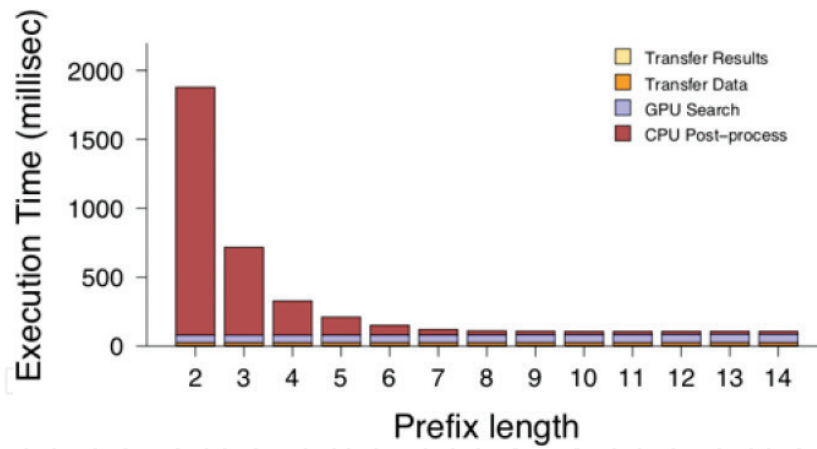
Figure 8.
Virus matching on the GPU.
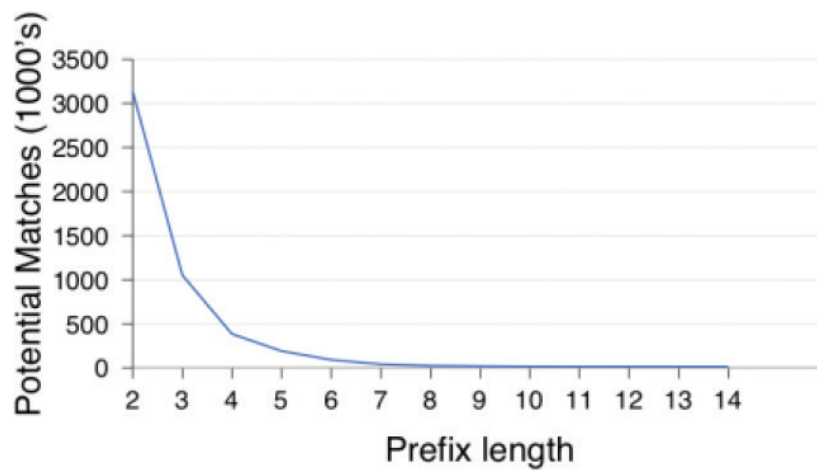
Figure 9.
Performance of GPU-assisted ClamAV and vanilla ClamAV. The performance number for ClamAV running on eight cores is also included. The CPU-only performance is still an order of magnitude less than the GPU-assisted.
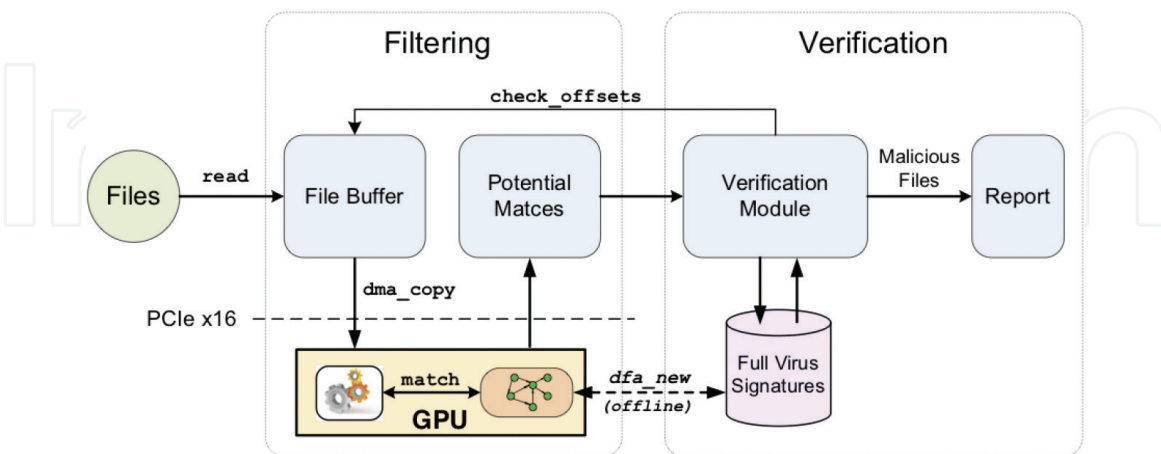
Figure 10.
Execution time breakdown.

throughput increases rapidly at a maximum of 20 Gbits/s. Moreover, the overall application throughput increases proportionally to the prefix length, due to the fact that the number of potential matches decreases, resulting to lower CPU post-processing. A plateau is reached for a prefix length of around 10 (**Figure 7**).

## 4. Kernel integrity monitoring

In this section, we describe how to leverage modern GPUs as a memory monitoring mechanism. In particular, we show the design of an external, snapshot-based, GPU-based kernel integrity tool that can be deployed in commodity servers and personal computers. Typically, a host memory integrity monitor should meet at least the following set of requirements [12, 13]: (i) Independence: the monitor needs to use a dedicated GPU that is completely isolated from the host. Our integrity monitor should operate continuously and detect any malicious action, notwithstanding the running state of the host machine. The GPU must be used exclusively for memory monitoring. Obviously, other usages can be served by any extra GPUs, if necessary, without affecting the proper usage of the kernel monitor. (ii) Host-memory access: the physical memory of the host must be accessed directly, in order to periodically check its integrity and detect any suspicious or malicious actions. (iii) Sufficient computational and memory resources: the system must be able to perform any requested computational operations and be capable to process large amounts of data efficiently. In addition, 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 host system. (iv) Out-of-band reporting: the system must be able to report the state of the host system over a secure channel. This can be achieved by establishing a secure connection that can be used to exchange valid reports, even in the case the host system has been fully compromised.

In order to meet the above requirements, several characteristics of the GPU's execution model require careful consideration. For instance, the typical GPU model in which a GPU kernel run for a while, perform some computations and then terminate cannot be considered secure. Instead, the coprocessor needs to execute in isolation, without being influenced by the host it protects. It is clear that leveraging GPUs for designing an independent environment with unrestricted memory access that will monitor the host's memory securely, is rather challenging. Many GPU characteristics must be considered carefully and in a particular way (**Figure 10**).

Figure 11 shows the overall architecture. In essence, the GPU continuously investigates, in terms of security, the specified kernel memory regions via DMA, over the PCI Express bus, and reports any suspicious or unwanted activity to an externally-connected admin station on the local network. From a high-level perspective, our system has two main parts that run in parallel: the device program (GPU code) and the host program (user process). The device program is responsible for continuously checking the integrity of requested memory regions and report any alerts. The host program periodically reads the status area and reports to the admin station, in the form of keep-alive messages. 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.

4.1 Autonomous GPU execution

Given that the host system may be vulnerable and could be compromised, it is important that the integrity monitoring of the operating system be completely isolated from the host, and guarantee that an adversary cannot tamper any code or data used by our system. Modern GPU chips follow a non-preemptive, cooperatively scheduled, execution style, which means that only a single kernel can run on the device at any point in time. As such, a bootstrapping method is employed that
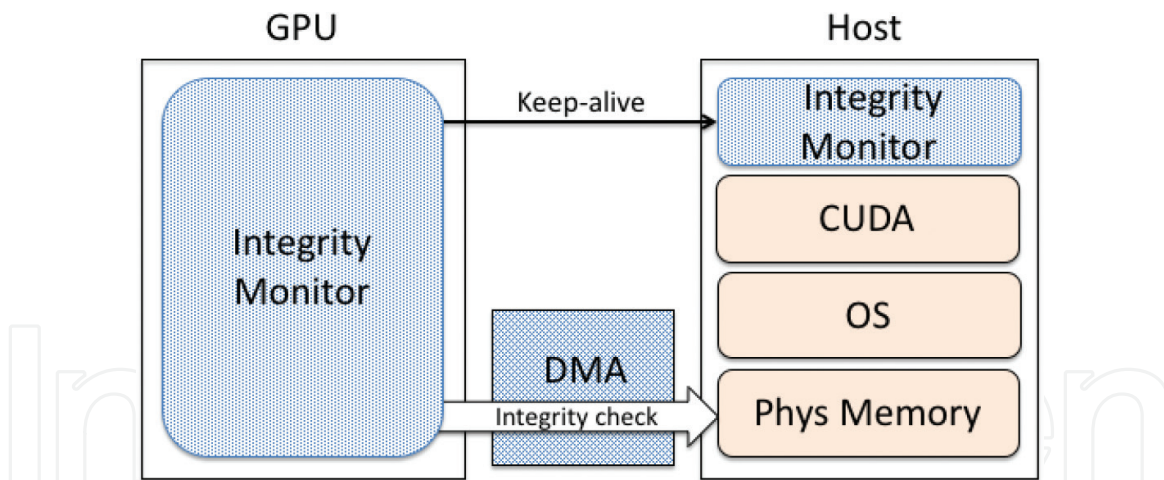
Figure 11.
The GPU-assisted integrity monitor architecture. The GPU continuously checks the integrity of host memory regions by accessing the corresponding device virtual addresses. To defend against man-in-the-middle and replay attacks on the reporting channel, a user program periodically sends an encrypted sequence number together with a status code to a separate admin station.

forbids any external interaction with our system. Any attempt to kill, pause, or suspend the GPU component of our system results in system shutdown, and the system can only resume its operation by repeating the bootstrap process. Even though many recent NVIDIA models conditionally support concurrent kernel execution, these conditions can be easily adjusted, by occupying occupy all resources. By doing so, no other, possibly malicious, code can run in parallel.

Even though these procedures ensure that our system can be initialized and run safely, current GPGPU frameworks (i.e., CUDA and OpenCL) do not support isolated and/or independent execution. To make matters worse, some drivers would even suspend a context in case the GPU kernel appears to be unresponsive. To overcome these issues, we configure the driver to ignore these types of checks. In addition, we create a second stream dedicated to the GPU kernel of our system, since 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. Therefore, all data communications with the host and the admin station are issued over a separate stream.

4.2 Host memory access

An important requirement for our GPU-assisted kernel integrity monitor is to implement a mechanism to access the requested memory pages that need to be monitored. Current GPGPU frameworks, such as CUDA and OpenCL, use a virtual address layer that is located within the host process virtual memory. Given that our system needs to access the kernel's memory, the kernel memory regions that are monitored should be mapped to the user process.

Typically, the access of memory regions that have not been assigned to a process is prohibited in the majority of modern OSes, including Linux and Windows. Every time a process accesses a page outside of its virtual address space, a segmentation violation will be thrown. To overcome such restrictions and be able to access the memory regions where the OS kernel text and data reside, we first need to map them to the user-space of the host process that has spawned the GPU kernel. To overcome the protection added by the OS, a separate loadable kernel module is used, which is able to map a requested memory region to the user-space. These memory regions can subsequently be registered to the GPU address space, using

the CUDA API. Afterwards, the GPU is able to access the requested kernel memory regions directly, through the physical address space, due to the fact that the GPU is a peripheral PCIe device (i.e., it only uses physical addressing to access the host memory). This feature enables us to un-map the user-space mappings of the kernel memory regions during the bootstrap phase, that would otherwise pose significant security risks.

4.2.1 Mapping kernel memory to GPU

During bootstrapping, our GPU-assisted integrity monitor acquires the kernel memory regions that need to be monitored. Given that these regions are located in the kernel virtual address space, the first step is to map them to the virtual address space of the user process, which spawns the execution of the kernel integrity monitoring GPU kernel.

In most modern operating systems, a peripheral device is able to bypass the virtual address layer and access physical addresses directly. The device driver creates a device-specific mapping of the device's address space that points to the corresponding host's physical pages. In order to map the corresponding kernel physical memory mappings to the GPU, we use a separate loadable kernel module that is responsible to provide the required page table mapping functionality. **Figure 12** shows the steps for mapping the OS kernel memory to the GPU. In step 1, the loadable kernel module resolves the physical mapping for a given kernel virtual address. In step 2, the kernel module allocates one page in the user context and saves its physical mapping; then it makes the allocated page point to the same page as the kernel virtual address by duplicating the PTE in the user-page table. Afterwards, in step 3, the kernel module maps this user page to the GPU (cudaHostRegister() with the cudaHostRegister-Mapped flag) and gets its device pointer via the cudaHostGetDevicePointer(). Last, the original physical mapping of the allocation is restored-and-freed by the kernel module (step 4). By doing so, any OS kernel memory page can be effectively mapped to the GPU address space. Furthermore, by un-mapping the user-allocated page right after the successful execution of the bootstrapping process, all intermediate mappings are destroyed. The same procedure is performed for all kernel virtual memory ranges that need to be monitored by our tool. The GPU driver populates a device-resident page table for the device in order to resolve its virtual addresses and spawn DMA transactions.

Finally, we compile Linux with the CONFIG_KALLSYMS_ALL = y and CONFIG_KALLSYMS = y flags, in order to have full access to the /proc./kallsyms interface. Even though this is not an inherent constraint for our design, it makes
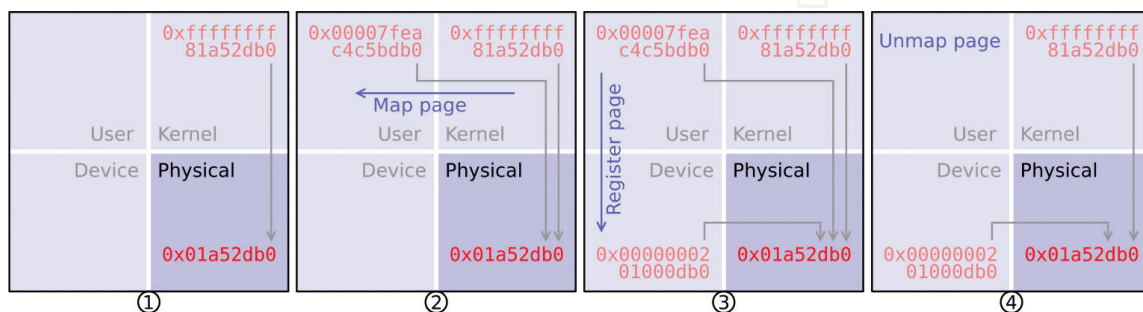
Figure 12.
Mapping OS kernel memory to the GPU. First, we have a kernel virtual address pointing to a physical address (step 1). In step 2 this mapping is duplicated to user space using a specialized kernel module capable to manipulate page tables. The user virtual address is then passed to a CUDA API call that pins the page into memory and creates the GPU-side page table entries (step 3). Finally, the intermediate user space mappings are destroyed (step 4), while the GPU continues to access the physical page.

the development and debugging much easier because it alleviates the need of custom memory scanners to search for the kernel page table address that need to be monitored. In case the access of the kernel symbol lookup table is not acceptable (i.e., in certain environments), we could locate the corresponding memory regions either using memory pattern scanners for dynamic loadable parts or via an external symbol table.

4.3 Memory integrity monitoring

The user can specify which host memory regions will be monitored; these regions can include, among others, pages that contain kernel text, kernel modules (LKM), and arrays or structures that contain function pointers (e.g., jump tables). Even though the hashing of static kernel text parts or loaded LKMs is straight-forward, still many parts of the OS kernel are quite dynamic. For instance, the data structures of the VFS layer change every time a new filesystem is mounted or unmounted. In addition, function pointers can be added dynamically by every loaded LKM.

As modern GPUs offer general purpose programmability, it is feasible to implement multi-step checks on different memory regions. These checks can be quite complex, such as for example in cases where several memory pointers need to be dereferenced in order to acquire a proper kernel memory address. Such type of checks can be supported by walking the kernel page table and resolving their virtual addresses dynamically from the GPU. Given 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.

Finally, we note that dynamic page table resolutions are not currently supported; instead, we need to provide a static list of kernel memory regions. Still, this is not an inherent limitation of a PCIe device, such as the GPU. The development of open-source frameworks (e.g., Gdev [14]) and drivers (e.g. Nouveau [15], PSCNV [16], etc.) will make the mapping of any physical page in the GPU address space feasible.

4.4 Sufficient resources

Modern GPUs are usually equipped with hundreds of cores and adequate amount of memory. This gives the ability to keep a sufficient number of memory snapshots and much state to detect complex, multi-step, attacks. It is clear though that these types of checks can become quite sophisticated, principally due to the lack of a generic framework that will give the opportunity to define detection modules on top of our architecture. Even though the support of such complicated memory checks is not supported currently by our GPU-assisted integrity tool, still we have tested the operation of aggressively reading and hashing memory, and as we demonstrate below, the GPU prevails the computational and memory resources to support them.

In particular, we measure the detection rate of a self-hiding LKM that is loaded, repeatedly, 100 times, using a different snapshot frequency. The self-hiding LKM acts similar to the operation of a rootkit, however it does not perform any malicious operations; instead, an actual rootkit will be exposed to the monitor for a longer time period, once loaded, in order to perform its malicious actions. The detection rate achieved is shown in **Figure 13**. We utilize a GTX770 under each configuration and use the CRC-32 for checksums (as defined by ISO 3309), due to its simplicity, speed, and its wide adoption. As can be shown, we can reliably detect (i.e., 100% detection rate) that a new kernel module has been loaded before hiding itself with a snapshot frequency of 9 KHz or more.

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. We notice that the memory alignment is major factor that significantly affects the performance of memory reads of the GPU; the most efficient memory reads are achieved with 16-byte aligned reads (or one uint4 in CUDA's device native data types). Unfortunately, since we cannot control how the kernel data structures will be placed in memory, we assume that many of our monitored regions will require one or two extra fetches.

In our final experiment, we focus on monitoring single memory pointer (8-bytes each). As shown in **Figure 14**, we can monitor at most 8 K pointers simultaneously without any loss in accuracy, due to the fact that we need to stay above the 9 KHz snapshot frequency. Obviously, this limits the amount of memory that we can monitor using our system, albeit 8 K addresses that are spread out in memory could potentially safeguard many important kernel data structures.
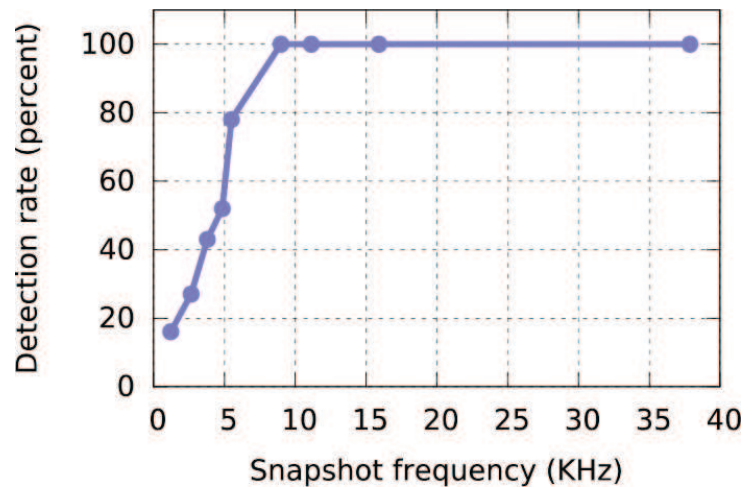
Figure 13.
Self-hiding LKM loading detection with different snapshot frequencies. For each case, a module is loaded that deletes itself from the kernel modules list 100 times, while monitoring the head of the list. A 100% detection rate with a snapshot frequency of 9 KHz or more is achieved.
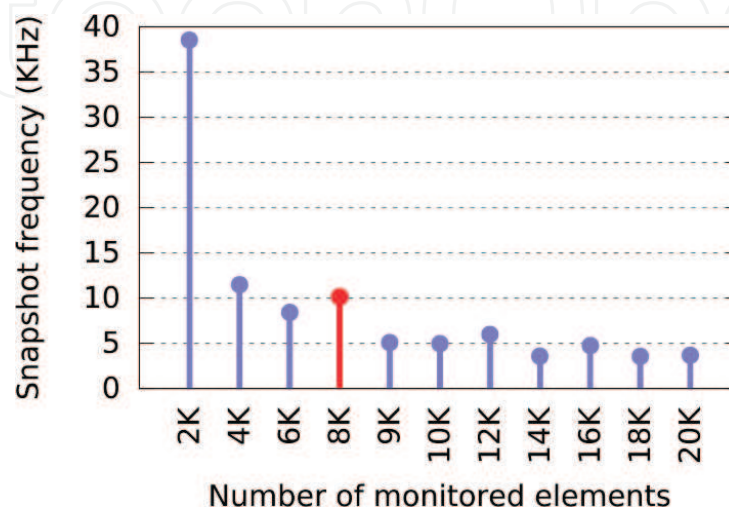
Figure 14.
Maximum achieved frequency according to the number of pointers being monitored. As the number of (8-byte) memory regions increases, the snapshot frequency decreases. A threshold of 9 KHz is required to accurately detect a self-hiding LKM loading, which is achieved with monitoring 8 K pointers.

4.5 Out-of-band execution

The coprocessor nature of GPUs offers limited defenses against itself. For instance, an attacker that has gained access on the host system can easily reset or disable the GPU device, and as a result, block its operation as integrity monitor. To solve this, an admin station needs to be deployed, completely isolated from the monitored system, that is responsible for keeping track of its proper state. Specifically, the user program that has spawned the GPU kernel that performs the integrity monitor, periodically sends keep-alive messages to the admin station via a virtual private connection. It is clear that simply sending messages to raise an alert is unsafe, due to the difficulty of the admin station to distinguish normal operation from a network partition or other failure. As such, we use keep-alive messages that encapsulate a GPU-generated status. In order to prevent attackers to send spoofed messages or replay old ones, we further encrypt the keep-alive messages together with a sequence number. Eventually, the secure channel between the admin station and the host is established at the bootstrapping phase. On the admin station, the reports are logged and is responsible that for the responsiveness of the monitor. Every time an alert is received or on the admin station, it takes the appropriate action. Moreover, the admin station takes care for any error case, such as missed reports (initiated by a time-out) or invalid messages.

Acknowledgements

Author details

Giorgos Vasiliadis
Foundation for Research and Technology—Hellas, Heraklion, Crete, Greece

*Address all correspondence to: gvasil@ics.forth.gr

IntechOpen

References

[1] Snort—Network Intrusion Detection & Prevention System. Available from: https://www.snort.org/

[2] Paxson V, Asanović K, Dharmapurikar S, Lockwood J, Pang R, Sommer R, et al. Rethinking hardware support for network analysis and intrusion prevention. In: Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HotSec). 2006

[3] Aho AV, Corasick MJ. Efficient string matching: An aid to bibliographic search. Communications of the ACM. 1975;**18**(6):333-340

[4] Norton M. Optimizing Pattern Matching for Intrusion Detection. Whitepaper. 2004. Available from: https://www.snort.org/documents/optimization-of-pattern-matches-for-ids

[5] PCRE: Perl Compatible Regular Expressions. Available from: http://www.pcre.org

[6] Thompson K. Programming techniques: Regular expression search algorithm. Communications of the ACM. 1968;**11**(6):419-422

[7] Berry G, Sethi R. From regular expressions to deterministic automata. Theoretical Computer Science. 1986;**48**(1):117-126

[8] Hopcroft JE, Ullman JD. Introduction to Automata Theory, Languages, and Computation. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1990

[9] ClamAV Open-source Antivirus. Available from: http://www.clamav.net/

[10] Boyer RS, Moore JS. A fast string searching algorithm. Communications of the Association for Computing Machinery. 1977;**20**(10):762-772

[11] Miretskiy Y, Das A, Wright CP, Zadok E. Avfs: An on-access anti-virus file system. In: Proceedings of the 13th USENIX Security Symposium. 2004

[12] Petroni NL Jr, Fraser T, Molina J, Arbaugh WA. Copilot: A coprocessor-based kernel runtime integrity monitor. In: USENIX Security Symposium. 2004

[13] Lee H, Moon H, Jang D, Kim K, Lee J, Paek Y, et al. KI-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In: USENIX Security Symposium. 2013

[14] shinpei0208/gdev. Available from: https://github.com/shinpei0208/gdev

[15] Nouveau driver for nVidia cards. Available from: http://nouveau.freedesktop.org/

[16] PathScale NVIDIA graphics driver. Available from: https://github.com/pathscale/pscnv