Using Commodity Coprocessors for Host Intrusion Detection

Mark M. Seeger

Thesis submitted to Gjøvik University College
for the degree of Doctor of Philosophy in Information Security

2012

Using Commodity Coprocessors for Host Intrusion Detection

Faculty of Computer Science and Media Technology
Gjøvik University College

Using Commodity Coprocessors for Host Intrusion Detection / Mark M. Seeger
Doctoral Dissertations at Gjøvik University College 1-2012
ISBN: 978-82-91313-94-8
ISSN: 1893-1227

To my parents.

Declaration of Authorship

I, Mark M. Seeger, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

(Mark M. Seeger)

Date:

Summary

The ever-rising importance of communication services and devices emphasizes the significance of intrusion detection. Besides general network attacks, private hosts in particular are within the focus of cyber criminals. Private data theft and the integration of individual hosts into large-scale botnets are two common purposes successfully subverted systems are used for.

In order to detect any attack, intrusion detection mechanisms need to probe the data in question. Therefore, the acquisition of sensor data is one of the fundamental steps in any intrusion detection system, as the execution of a detection algorithm – be it anomaly- or signature-based – relies on the integrity of the assessed data. In cases where the intrusion detection system (and the sensor data acquisition component, in particular) is installed on the very same host it is supposed to protect, attacks against its preventive and detective safeguards are rather simple and supported by potential vulnerabilities of the host's operating system.

Detection speed plays a vital role in keeping the damage caused by subversion attempts as small as possible. Dispatching the data acquisition and detection mechanisms from the host is desirable, as a higher degree of independence allows high-speed execution even in cases where the host has already been infected, or where its central processing units work to capacity.

The history of computer science, with cryptography being an excellent example, has taught us that the level of security can be increased by outsourcing certain operations to additional, special-purpose hardware. Here, a positive side effect is that the increase in security is often accompanied by an increased speed at which the corresponding operations can be executed.

The present thesis seizes upon the idea of outsourcing, but rather than employing additional special-purpose hardware, it proposes the execution of relevant operations on commodity hardware. While the application of coprocessors for network intrusion detection is common practice, and approaches using PCI add-in cards, as well as external cryptographic coprocessors exist, we propose the application of commodity coprocessors for host intrusion detection, i.e., modern graphics processing cards (GPU) found in current laptop and desktop computers.

Our focus was on validating the assumption that modern GPUs are, in general, applicable in the task of acquiring host sensor data for intrusion detection purposes. Thus, we propose their application as independent auditors, and present research results regarding their feasibility to function as such.

We detail abstract cost models and their practical validation, as well as a proof of concept implementation of an autonomous GPU kernel. This allows us to conclude that – leaving aside their programming and runtime frameworks – commodity, off-the-shelf coprocessors (i.e., modern GPUs) are able to perform host observation tasks in an unintrusive manner.

Acknowledgments

I dedicate this thesis to my parents, Agnes and Günter Seeger. From many points of view, they are the ones who made this work possible. They never imposed their ideals on me, nevertheless giving me guidance whenever I needed it. They let me become the person I am today by giving me the freedom of a bird combined with tremendous support whenever required – for better and for worse. I will never be able to give back as much as I was given. Thank you!

Sascia, I want to take this opportunity to thank you for who you are and for being within the core of my heart. I cannot imagine my life without you anymore. You are my living proof, that for every puzzle piece, there is a matching counterpart.

Special appreciation goes to my first supervisor Stephen D. Wolthusen who clearly contributed the most important scientific guidance which enabled me to finally write this thesis. His style of supervision clearly shaped me as a researcher, and my thesis is manifesting evidence for this. He always impressed me with his technical knowledge and his overarching overview of relevant literature. Despite working to capacity, he agreed to supervise me and often pointed me in the right direction. Every time I sent him early versions of my writings, he was able to comment on the essence, to encourage me to take deeply technical specifications into considerations, as well as to assess adjacent approaches concerned with a related domain.

I also want to thank Christoph Busch, my second supervisor, for the time he took to comment on all of my publications and for the discussions we had. His commitment gave me the possibility to spend research periods abroad and to participate in several international conferences.

No doubt, without the support of Stephen and Christoph, this thesis would not have been possible. Nevertheless, it was Klaus Kasper who introduced me to Christoph in 2008 and made me aware of the newly established CASED. Thus, it is somehow by his merit that I had the chance to apply for the position of a doctoral researcher and that I eventually became part of this aspiring research cluster.

I would also like to thank Peter Herrmann, Thomas Kemmerich, and Katrin Franke for being part of the examination committee, and Patrick Bours for being the head of the committee.

For their excellent practical support, credit goes to my student assistants, Julian Knauer and Pierre Schnarz: thank you, guys.

Naming all people who have come along with me during my time as a Ph.D. student would clearly break the mold, and, therefore, I apologize to those who have not been named explicitly. In this context, I want to express my appreciation to the Gjøvik University College (GUC), Norway, the University of Applied Sciences Darmstadt (h_da), Germany, and the Center for Advanced Security Research Darmstadt (CASED), Germany for their administrative, technical, and financial support throughout the past three years.

Mark M. Seeger
August 22, 2012

Contents

List of Figures

List of Tables

List of Listings

Part I

Inception

Chapter 1

Introduction

> Es kommt nicht darauf an, mit dem Kopf durch die Wand zu rennen, sondern mit den Augen die Tür zu finden.
>
> WERNER VON SIEMENS

1.1 Motivation

Currently, modern commercial host intrusion detection systems (IDS) are installed on the host system itself. That is, they operate next to all other software, are executed by the host system's central processing units (CPU), occupy parts of the host system's hard disk drive (HDD) and system memory (i.e., RAM: random access memory), and consume CPU cycles. In spite of these characteristics, the most severe fact is that they rely on the security of the underlying host operating system (OS), which makes them highly assailable. By exploiting host OS vulnerabilities, a host IDS can be shut down and scan results can be tampered with. With ready-to-use exploits, included in software products such as the Metasploit Framework, a free penetration testing solution, even moderately talented adversaries are given the opportunity to execute targeted subversion attempts against individual host systems.

No matter what kind of intrusion detection system is used (i.e., signature-, specification- or anomaly-based), they all rely on the integrity of sensor data. Incomplete or falsified data make it difficult, if not impossible, to detect even simple attacks. Therefore, it is of paramount importance to ensure the integrity of sensor data, as well as the component that is in charge of acquiring this data. Here, the application of external hardware, i.e., coprocessors, is one possible and often-applied defense against traditional host system exploits. Due to the frequency at which an observation mechanism could be operated when running on an autarchic coprocessor, subversion attempts would be detectable while they are ongoing, giving the host observation system the chance to intervene before the attack is completed.

In general, a coprocessor is any processor assisting the CPUs. While in the early days such processors were mainly used for mathematical computations (e.g., ALU: arithmetic logic unit), the concept has advanced over time, and is ubiquitous nowadays in all types of computer architecture, ranging from mobile phones to mainframes. While most coprocessors are developed to perform very limited tasks much faster than the CPUs could, they are often also limited in terms of programmability, which prevents their use in other operations. Here, application-specific integrated circuits (ASIC), Field programmable gate arrays (FPGA) and modern graphics programming units (GPU) present common exceptions. These coprocessors offer a programming interface which allows the implementation of custom logic and algorithms. While ASICs and FPGAs are peripheral hardware, freely programmable GPUs are considered commodity hardware with respect to modern desktop and mobile computers.

Today, GPUs are high performance processors, distinguished by a massively parallel hardware architecture, and, due to the existence of abstract programming and runtime

frameworks, possess an applicability which reaches far beyond the graphics processing scope. To this extent, the two leading GPU vendors NVIDIA and AMD/ATI are dominating the market with CUDA and ATI Stream, respectively.

Besides graphics computations, which may still be their largest field of application, GPUs are commonly used today for non-graphics computations whenever processor-intense calculations are required. Common examples are financial engineering, weather forecasting, bioinformatics and with respect to information security cryptography and intrusion detection. For intrusion detection purposes, GPUs, and coprocessors in general, are generally only applied in the field of network intrusion detection. Here, network traffic is assessed by algorithms executed on the GPU in order to keep latency to a minimum. The field of host intrusion detection has not yet witnessed the broad-based application of coprocessors.

Having such a coprocessor perform host intrusion detection would eliminate several drawbacks of traditional systems, and would additionally create new opportunities in the ways we defend personal computers from attackers in the near future. The parallel architecture has more to offer than pure superiority of speed over the host's CPUs, which is nonetheless an advantage. Due to its design, the GPU can be seen as an autonomous and concurrent coprocessor, having autarchic physical memory and, of course, its own processing units. These design features can be used to build a host intrusion detection system decoupled from the host while still able to perform observations in real time. With no references to host APIs (application programming interface) or DLLs (dynamic link library), there would be no intuitive way to tamper with such an observation mechanism by, for instance, exploiting host OS vulnerabilities. Furthermore, if intrusion detection obligations no longer burden the CPUs, more computing power is at the disposal of the user.

1.2 Research Questions

With sensor data acquisition being one of the very fundamental techniques of any observation system, our research focuses on the application of modern GPUs for this purpose, in the field of host intrusion detection. Since GPUs are powerful coprocessors that combine the tamper resistance of external auxiliary hardware with the flexibility, programmability, and ubiquity of internal standard hardware, using them for security related tasks seems promising.

To this extent, our main research question can be formulated as follows:

> *What are the limitations of an asynchronously and concurrently operating commodity coprocessor when applied for efficient observation of a host's system state, and how can these limitations be overcome?*

The goal of this Ph.D. thesis is to provide a theoretical, as well as practical validation of the assumption that commodity coprocessors (i.e., the GPU) can be used for host intrusion detection purposes.

The GPU distinguishes itself from onboard coprocessors in terms of applicability and computational power, and from external auxiliary processors such as FPGAs in terms of host connectivity and the degree of its availability with respect to desktop computers and laptops. These characteristics theoretically make it an ideal off-host host observation processor. As its primary application lies in graphics computations, there is currently no intuitive way to create a GPU-host connection that authorizes an unrestricted data access. Therefore, observation of a host system in a thorough and efficient way is not possible using the graphics processor *as is*.

Even if a host observation mechanism was operated in total self-reliance, without any host dependent software modules, resource contention on the host-side would take place, due to forced synchronizations between the host's system memory and its caches. Model-

ing the degree of performance degradation on the host-side, followed by a practical validation, was subject of our research.

Our ultimate goal was to validate our assumption that modern GPUs are in fact suitable for performing host observation tasks. While we also assumed that off-the-shelf programming and runtime frameworks will prevent total independence from the host, we investigated the degree to which unmodified GPUs allow autonomous, as well as concurrent execution, and we compared existing approaches and their properties with ours.

Novel techniques operating on modern hardware often allow faster and more precise detection of known and possibly unknown attacks. This does by no means allow the inference that existing and well-studied threats are automatically rendered harmless. We were, therefore, also interested in the nature of attacks that could not be detected even if the observation mechanism was to run on a high performance coprocessor.

With fast and reliable data acquisition and observation mechanisms at our disposal, we also focused on how sampling strategies in combination with the observation of crucial measuring points can lead to faster detection speed, as well as possible further advantages over computer criminals trying to subvert host computers.

The above briefly motivated research scope is reflected by the following sub research questions (RQ).

RQ1: What performance impact results from initiating observations of a host system's shared memory by a commodity co-processor?

- **1.1:** How can we model the loss of performance?
- **1.2:** How can we practically measure the loss of performance?

RQ2: What architectural features of modern GPUs counter their usability as independent auditors?

RQ3: What is the nature of attacks for which the general advantages of GPUs render less powerful?

RQ4: Presuming a high performance, asynchronously- and concurrently-operating, host- observing commodity coprocessor: How can sampling strategies facilitate real-time detection?

We will now present the answers to the above-stated research questions and point to the publications in which each of them is regarded in detail, before summarizing each contribution separately in Section 1.3.

1.2.1 RQ1: What performance impact results from initiating observations of a host system's shared memory by a commodity co-processor?

The detailed answers to this question can be found in Seeger and Wolthusen [10] and Seeger et al. [13].

In order to provide a qualitative answer to this question, we inspected the architectural details of modern computer systems. That is, we focused on features of non-uniform memory architectures (NUMA) in order to describe a model capable of expressing the performance degradation of a certain process when it is subject to frequent observation. We discovered that, in particular, techniques that act as performance boosters under *normal* conditions are the ones that cause performance penalties in the case of (coprocessor) driven observation. To be exact, snooping protocols, write-back and working-set strategies are implemented for the general case of data consumption by the host's CPU. Data accesses according to the busy-wait approach operated by a processor other than the CPU cause these protocols and strategies to act as performance bottlenecks. Furthermore, it is the composition of the data under observation that yields information about the degree of performance degradation. While observing read-only data causes no synthetic write-backs,

reading writable data actually being written confuses working-sets and causes forced synchronizations [10].

The quantitative validation of the proposed model is given in Seeger et al. [13]. We showed that the performance degradation is quantifiable but depends on many parameters. Due to architectural features of the system under observation (e.g., number of CPUs, enabled power saving features, cache size, write-back strategy, etc.), and the fact that there is no such thing as a general data access pattern for real-life applications, the extent of degradation may vary. In an experimental setup, we experienced a performance loss of more than 25% when we observed two customized processes in parallel, each of them working on a 64 byte data structure.

The two presented models and their validation reveal that relieving the host's CPUs from host intrusion detection duties does not imply that the performance loss is fully compensated. In fact, due to main memory accesses from off-host coprocessors, combined with the existence of internal cache and snooping protocols, forced synchronizations take place inside the host's memory hierarchy, causing performance degradations.

1.2.2 RQ2: What architectural features of modern GPUs counter their usability as independent auditors?

The detailed answers to this question can be found in Riedmüller et al. [8] and Seeger and Wolthusen [12].

The development of an off-host host intrusion detection mechanism operated by the GPU is possible, but – due to architectural constraints – does not offer the required level of security. By using an unmodified GPU along with its programming and runtime framework, only predefined parts of the host's physical memory can be accessed and the detection mechanism is highly dependent on host-side software components. We investigated the architecture of a modern GPU and compared its features to a set of properties (cf. [4, 7]) that must be fulfilled in order to operate an observation mechanism as an independent auditor. We discovered that only three of the ten criteria are met: Inaccessibility, physical secureness, and the provision of sufficient processing power. On the other hand, properties such as the verifiability of the software operating the GPU, the lack of non-volatile memory, or the fact that unrestricted access to the host's system memory is not provided, counter a GPU's off-the-shelf usability as an independent auditor [8]. Next to their parallel architecture, it is the much smaller instruction set that causes the immense performance boost compared to CPUs. On the other hand, this limits their applicability in terms of computational sophistication, and hinders them in executing, for instance, code compiled for x86 architecture.

With respect to these results, restriction in accessing the target's memory is superior, as full data access forms the base of any detection system. To this extent, we have validated the assumption that access restrictions can be bypassed as the DMA (direct memory access) connection between GPU and host generally allows the access of any addressable data [12]. We developed a GPU kernel that operates without host-side runtime references. Given a proper configuration, as well as manual initialization of the device and the kernel as such, we were able to execute simple operations in total independence of the host. In this case, all control lies within the GPU itself. Moreover, due to a lack of logical host connectivity, the observation mechanism becomes, in principle, *invisible* to an attacker, and if all software parts are solely stored on the GPU itself, it is immune to host OS vulnerabilities and can function properly even when the host system has already been subverted.

1.2.3 RQ3: What is the nature of attacks for which the general advantages of GPUs render less powerful?

The detailed answers to this question can be found in Seeger [9].

Using an off-host coprocessor to detect computer attacks is a very promising technique. Once the technical constraints that burden current GPUs and their frameworks are overcome, GPUs have the power to operate a host intrusion detection mechanism distinguished by strong tamper resistance, high performance, and low overhead. Nevertheless, even off-host host intrusion detection systems will not be immune to certain types of known threats. This becomes clear when taking our proof of concept implementation of a dynamically altering control-flow approach into consideration. By analyzing program code and alienating a Linux system call mostly used for breakpoint debugging, we were able to overwrite specific variables stored on the heap. As these variables were marked for runtime alternations, it is hard to tell whether a change in value is benign or not. Thus, detecting such attacks is less of a technical problem than it is an architectural one. If the system design allows the alternation of certain variables during runtime, no purely technical approach will be able to tell legal alternations apart from illegal ones. This is, of course, also true for coprocessor driven solutions.

1.2.4 RQ4: Presuming a high performance, asynchronously- and concurrently-operating, host- observing commodity coprocessor: How can sampling strategies facilitate real-time detection?

The detailed answers to this question can be found in Seeger and Wolthusen [11].

Modern cyber attacks often consist of several phases, such as the planning and reconnaissance phase, which take place before the actual attack. From a technical point of view, each attack itself is confronted with inevitable dependencies, where a certain step can only be accomplished once its prerequisites have been fulfilled. Presuming the targeted observation of critical data structures, we presented a causality model built upon the potentially caused relation proposed by Tarafda and Garg [14]. This model is based on probabilistic assumptions regarding the causal dependency of attack sets, and can be used to tune observation strategies to be more target-oriented. The overall observation frequency can be adjusted (i.e., lowered) as we take into account that certain data elements can only be attacked once certain activities have been observed. By only reducing the detection probability by 1% in our experiment, the burden of observing all data elements with the same frequency could be absorbed. Furthermore, the proposed model allows the triggering of counter measurements even before the attack succeeds.

1.3 Summary of Thesis Contributions

Each publication and its contributions is summarized in this section. All publications are presented in the order they were published, starting with the earliest.

As already reflected by research question RQ1, the possible performance degradation in the host under observation was one of our core focal points. In our models, presented in [10] (cf. Chapter 3), we regard the host's physical memory as the connection between an off-host coprocessor and the host itself. Therefore, any coprocessor-based host observation must inevitably access data stored within this memory entity. On the host itself, the RAM acts as a link between caches and hard disk drives. Data currently consumed (or likely to be consumed in the near future) by the CPUs are stored here and replicated throughout the different caching hierarchies. Due to performance considerations, data altered in the highest levels of these hierarchies are not instantly written back to the lower levels. Besides different protocols that take care of data consistency, write-backs are forced whenever outdated data in the RAM is accessed by a different processor. That is, in order to serve any

accessing process with the latest data, altered data from the first level cache is written back to the RAM, causing a wait situation for that process.

The contribution of [10] is the proposal of two cost models. The first cost model is built upon a simplified model presented by Hennessy and Patterson [3]. It allows the expression of the synchronization cost, not only for one caching hierarchy, as presented in [3], but for multiple such hierarchies.

To this extent, we differentiate between demand-related and coherence-related misses, which represent the penalty times for the demand for data that needs to be retrieved from a lower (demand-related) or adjacent (coherence-related) memory level. The second cost model focuses on the composition of sets of most recently referenced pages[1], referred to as working sets [1], and the interference that can be caused when these sets are indirectly accessed by a coprocessor. In theory, if all resources a certain process needs were to fit into one working set, and if this set was to be consumed by one process only, demand-, as well as coherence-related misses would be eliminated. As this is very unrealistic for real-world applications, both types of misses occur continuously. Any sort of artificial interaction with the working set (e.g., caused by forced synchronizations due to coprocessor-based host observation) has the potential to increase the interference frequency. In order to describe the possible severity of this interference, we introduced the interference ratio I, which takes the composition of the working set into account in terms of read-only and writable parts, as well as the percentage of processing time actually used to write to the writable pages. This model proposes that observing read-only data and writable data that is, in fact, not being written causes much less performance degradation on the host-side than observing writable data being written does.

In order to practically validate our models as they are proposed in [10], we first assessed modern GPUs with respect to their feasibility to act as so-called *independent auditors* in [8] (cf. Chapter 4). This assessment was necessary in order to answer research question RQ2. The reason for this is that today's GPUs are powerful commodity hardware and are widely available in computer systems of all sizes. Due to their system architecture, they can be regarded as the most powerful and most widely spread coprocessors for personal computers. Ever since the appearance of vendor-dependent programming and runtime environments (i.e., NVIDIA CUDA and AMD/ATI Stream), GPUs can be easily applied to tasks beyond the graphics computations' scope. Being both powerful and ubiquitous makes these coprocessors profitable operators of off-host host observation systems.

The contribution of [8] is the assessment of the combination of modern off-the-shelf GPUs with their corresponding frameworks with regard to their ability to operate a sustainable off-host host observation system. The assessment, according to properties of existing coprocessor solutions using extraordinary hardware such as PCI (peripheral component interconnect) add-on cards [4, 7], revealed severe shortcomings that cannot be easily circumvented. Out of the ten proposed properties, only three are fulfilled by GPUs. These are inaccessibility, meaning that the host must not have access to the internals of the GPU, physical secureness, and the provision of sufficient processing power. The properties not fulfilled by GPUs are unrestricted memory access, provision of secure transactions, continuity, transparency, verifiability, provision of non-volatile memory, and the ability to provide reports in a secure manner. In addition to the assessment according to the provided properties, we revealed the following drawbacks: If GPUs and their frameworks are used *as is*, they rely on host-side APIs, DLLs, and software components. Terminating the references to these components terminates the kernels executed by the GPU. Additionally, GPU kernels can only be started through software components running in the user space of the host system, which makes them easily attackable. Furthermore, a maximum GPU kernel runtime is traditionally set. While this seems useful in preventing system hangs, it implies a severe weakness with respect to intrusion detection purposes.

[1] The term *pages* can be replaced by any other unit of data.

In order to use a GPU as an independent, asynchronously, and concurrently working auditor, at least the following architectural modifications must be made: It must be possible for the GPU to have unrestricted access to the host's physical memory; the observation mechanism must be started on system boot (i.e., before the host OS is started); there must be no logical connection to host-side software components such as APIs, DLLs, or any other kind of host-side software, and at least a small amount of non-volatile on-device memory must be provided.

Even though the results of our GPU assessment, presented in [8], prevent off-the-shelf applicability of modern graphics cards in functioning as autonomous auditors, one of our goals was to precisely quantify the performance degradation on the host-side when off-host observation takes place. What has been theoretically validated in previous work [10] needed practical confirmation. This confirmation is the subject of [13] (cf. Chapter 5).

Therefore, the contribution of [13] is the quantification of the maximum performance degradation caused by observing a self-implemented, and, therefore, rule-governed process, using an off-host coprocessor. By adjusting the parameters of the process to be observed, we were also able to practically validate the cost model proposed in earlier work [10]. Due to the constraints revealed by Riedmüller et al. [8], the experimental setup was not comprised of a GPU and a computer, but by two computers, one being the target and one being the observer. Having a standalone computer act as a coprocessor of another computer may not seem feasible in many real-world scenarios, including host intrusion detection. Despite this, it serves the purpose of fulfilling the properties of a self-sufficient auditor (cf. [8]). The two computers were connected using FireWire technology, which, due to a design feature of the open host controller interface (OHCI) implementation, allows full access to the target's physical memory (cf. [6, Chapter 12]). This feature provided the possibility of observing a prepared process on the target, which, in turn, recorded the CPU cycles consumed for its duties. By having the target execute predefined tasks with and without observation, we were able to quantify the performance degradation due to observation.

With respect to the cost model proposed in earlier work (cf. [10]), we adjusted the observed process in a way that only a portion of its writable data was actually being written. While the general value of the cost model was confirmed by the gathered results, the information gleaned from the experiments has led to a slightly revised version of the model (cf. Equation 5.1 of Chapter 5). Despite the minor adjustments made, the message remained the same: Read-only data cannot be written, therefore, does not need to be synchronized, and thus will not cause resource contention due to forced synchronizations.

Across all experiments, a maximum performance degradation of more than 25% was measured for the setup in which we observed two of our rule-governed processes, each of them working on a 64 byte data structure. In our test scenario, this must be interpreted as the worst case, as 100% of all 128 bytes were writable and, in fact, being written. With respect to the extent of the performance degradation caused by the observation mechanism, we also must take the fact that the connection between observer and target (400 Mbit/s) was rather slow compared to the speed of the GPU-host connection (up to 16 GB/s) into account. Therefore, the performance degradation caused by applying a GPU would be much higher if following the busy-wait approach.

In addition to the performance details regarding our coprocessor-based approach for host observation, the identification of the nature of threats that, despite the promising properties of an off-host coprocessor, cannot dominate, has also been examined. The corresponding results of the analysis are presented in [9] (cf. Chapter 6). To achieve this goal, we focused on flow-analyzing techniques that are commonly used in the field of information security in order to uncover subversion attempts based on the alteration of program flow. Nowadays, buffer-overflow attacks are the most widely-spread examples of exploiting the deviation of control-flow. Here, software vulnerabilities are used in order to inject and execute defective code. Most – if not all – approaches to guarding a program's control-flow

rely on a static analysis, which is, as among others demonstrated by Moser et al. [5], no longer a sufficient technique for identifying malware.

The contribution of this paper is the investigation and comparison of three common prototypes in the field of control-flow analysis, and the presentation of a dynamic control-flow attack, which neither commercial anti-virus suites nor the assessed tools could counter or detect. Two of the three prototypes we assessed rely on the output quality of 3^{rd} party tools, which they build upon, and all three tools operate statically, and thus have one shortcoming in common: They are not designed to instantly react to changes in the executional flow caused by dynamically-loaded code. To this extent, we have developed a C++ program which is capable of dynamically altering the control-flow of arbitrary executables by using a late binding technique for loading dynamic link libraries (DLL). Because this approach does not exploit any soft- or hardware vulnerability, it stays undetected. Detecting such unwanted changes in control-flow confronts the counter measurements with the problem of distinguishing legal alternations of the values of variables stored on the heap from illegal ones. Or, to cite Forrest et al. [2], "to distinguish *self* from *other*". This is a very general problem, and the inability of identifying such attacks is indeed a problem not unique to current state-of-the art commercial intrusion detection suits. It was no surprise that the proposed observation and sensor data acquisition approach was unable to detect any attack of this kind, as the subject of the present thesis is not the proposal of a novel general intrusion detection system. Instead, based on and motivated by the results presented in [8], we succeeded in validating that modern GPUs *can* be used to autonomously and concurrently perform host observation in cases where host-side programming and runtime frameworks are not used.

Clearly, the use of an off-host coprocessor to perform host intrusion detection can have various motivations. One major argument stems from the fact that even moderately sophisticated attacks will target audit data collection and security controls (i.e., preventive and detective safeguards). Achieving this is relatively easy when the corresponding mechanisms operate in a highly heterogeneous software environment, as complex software tends to not be fully flawless, giving adversaries the chance to take advantage of said programming shortcomings. Security holes in prominent software are constantly patched while new vulnerabilities are discovered at the same time. Therefore, it is necessary to investigate modern ways of performing host intrusion detection. This includes the application of modern techniques such as machine learning, but also requires the improvement of traditional methods (i.e., signature matching) for which an equivalent substitute has yet to be found.

To this extent, the contribution of [12] is a presentation of a proof of concept implementation of an autonomous GPU kernel able to perform concurrent host observation tasks. As revealed by Riedmüller et al. [8], modern GPUs cannot intuitively be used as autonomous auditors, mostly due to the constraints of the corresponding runtime frameworks. By leaving the standard GPU programming model aside, we developed a mechanism through which the execution of code on a GPU is allowed without the host being actively involved. That is, in a proof of concept implementation of an autonomously running GPU kernel, we validated our assumption that a GPU can execute code even after the initializing process is terminated. This was achieved with proper hardware configuration and a manual initialization of the device as such, as well as the context in which the kernel runs. Once the kernel has been started, we forced the starting process to exit and observed the memory region on which the kernel operated from a second application. While the kernel itself performed rather simple read/write operations in this proof of concept, enriching it to the extent to which it can actually perform security related tasks is simply a matter of good programming skills.

In addition to detailing the implementation as such, we discuss its security aspects and present the results of a brief performance benchmark. The untuned version of our sensor data acquisition component was able to operate with a maximum transfer rate of about 1

GB/s.

This result once again shows that one advantage coprocessors hold over the CPU is their dominance in terms of speed, which also incorporates a severe drawback if applied in conjunction with an unfavorable sampling strategy. That is, in [13], we quantified the performance degradation in an experimental setup and reported a performance reduction of more than 25% in one case. While we used a rather slow coprocessor/host connection (i.e., FireWire) for this puspose, it is easy to imagine how severe the performance degradation can become when using a high performance coprocessor such as the GPU. Therefore, achieving a high observing frequency is straightforward, while the busy-wait approach (i.e., maximum frequency) is undesirable.

With respect to our last research question (i.e., RQ4), the contribution of [11] is the proposal of essentially one model, which can be used to take attack characteristics into consideration in order to optimize the observation strategy of a coprocessor-driven host intrusion detection system.

The operation of detection algorithms on a snapshot of data is exposed to the risk that malicious activities are ongoing while the acquired data is assessed. Therefore, limiting the observation mechanism to a set of comprehensive data elements, while at the same time recognizing the possibility of causalities between said elements, seems promising in achieving the twofold objective of providing a reasonable level of security and causing as little performance degradation as possible.

The proposed model does not only take causal dependencies between so-called attack sets into account, but also incorporates the likelihood of observations being suppressed by other operations of higher priority, as well as the possibility that an observed alternation may have a benign cause. It is worth mentioning that the latter feature may be applied in the context of dynamic control-flow detection as well, as discussed in [9].

By applying partly-altered versions of well-known models in order to incorporate our assumptions, we experience a negligible reduction of 1%, in terms of detection probability. In contrast to existing approaches, our model demonstrates a significant increase in efficiency, as neither bulk data snapshots have to be taken nor are we required to observe all data elements with the same (possibly high) frequency. Additionally to that, the model reveals the probabilistic chance to be one step ahead of the attacker. That is, given the assumed causal dependencies, a proactive action, for instance, locking the concerned data element, can be undertaken.

With respect to the ability to perform real-time observation, the practical application of the causality model showed that the twofold objective of performing unsuppressed observations while having a minimum failure rate (cf. Section 8.4 of Chapter 8) can be achieved by taking advantage of the coprocessor's computational dominance. As modern GPUs continue to maintain performance superiority, these results underpin the initial proposal for the use of commodity coprocessors in host intrusion detection made in 2010 (cf. [10]).

1.4 Conclusions and Future Work

Current intrusion detection systems are installed on the very same system they are supposed to protect. They consume valuable system memory and CPU cycles, and are subject to rather simple attacks, as they rely on the security of the underlying host operating system. Consequently, an intruder can easily turn them down or may tamper with the sensor data acquisition module as such. While the former case can be detected and signaled to the end user, the latter may stay undiscovered, as the intrusion detection component cannot find any threat if the data given to it by the host sensors does not show any inconsistencies.

Ideally, a host intrusion detection system would be an autonomous, concurrent, and self-sufficient soft- and hardware component with full access to the host system's memory. As will be presented in depth in Chapter 2, Related Work, various approaches have been proposed to achieve this ambitious goal.

The present work proposes the application of modern graphics programming units to this extent, as GPUs combine the autonomy of external auxiliary hardware with the availability of commodity hardware. Due to their small instruction set and parallel architecture, GPUs are able to perform dedicated tasks at a much higher speed than CPUs. Therefore, our first scientific contribution was a proposal of two cost models in 2010 [10]. With the assistance of these models, the performance degradation caused by applying coprocessors in host intrusion detection can be determined. Under the assumption that, not only the size of the data to be observed, but its constitution in terms of read-only, writable and writable pages actually being written matters, the general loss of performance of processes under observation can be identified. The practical validation of this model, as well as the quantification of said performance loss was the contribution of [13]. Here, we configured a second desktop computer to act as a coprocessor of another one and established an interconnection between the two, taking advantage of a design feature of the FireWire technology. We gained accurate performance benchmarks from an observed process which reported the CPU cycles consumed in both cases; with and without being under observation.

In [8], we assessed the architecture, i.e., programming, and runtime frameworks of modern GPUs, and compared the corresponding features against given properties of autonomous host auditors. The assessment revealed that the architecture of modern GPUs is generally suitable for building observers that suffice the required properties of self-sufficient auditors. It is, therefore, the current software frameworks of these commodity coprocessors that are limiting their application for secure and fully asynchronous host observations.

By waiving the aforementioned frameworks, we were able to manually configure a GPU kernel and have it run on a GPU, even after the initializing process has already been terminated. This provided validation for the results presented in [8] and was the contribution of Seeger and Wolthusen [12].

Coprocessor-based host intrusion detection may be an answer to many of today's questions concerning the security of a host intrusion detection system, but, indeed, is not the answer to every threat. To this extent, we have developed a proof of concept application that is capable of dynamically altering a program's control-flow by taking advantage of the late binding technique. As this technique overwrites references stored on the heap, detecting such alterations is straightforward, while the automatic distinction between legal and illegal write accesses remains problematic. These results, together with an evaluation of three common prototypes focusing on control-flow integrity, are the contributions of Seeger [9] and present one example of the nature of attacks for which the GPUs intrinsic characteristics do not dominate existing host-bound approaches.

Our latest contribution can be found in Seeger and Wolthusen [11]. Here, we propose a causality model which allows for the reduction of observation frequency through the consideration of causal dependencies between critical host-side data elements and attack sets. Besides allowing a targeted observation, the probabilistic results gained from the model also give an observation mechanism the chance to be one step ahead of the attacker. That is, under the assumption that a first step always precedes a second one, we can eventually prevent the second step from being successful once we have seen the first.

One obvious question that has not been addressed so far concerns the acquired sensor data. We have presented that gathering such data from the host using a GPU is possible at the expense of performance, in cases where the sampling rate is too high. What has not been addressed so far is the question of what to do with this data. Depending on how complete the data samples are, and keeping the tremendous computing power of modern GPUs in mind, it would be possible to calculate a graph of all possible states between sampling point α and β. While this requires knowledge about valid system conditions, it also appears to be a promising approach to revealing plausible and consistent, but nevertheless counterfeited, states.

In general, it would be interesting to analyze existing intrusion detection algorithms

to find out how well they can operate on our sensor data. This should include signature-based, anomaly-based, and specification-based algorithms, likewise. To elaborate the degree to which an algorithm (or a class of algorithms) can deal with missing data points, could be a starting point. The ability to operate on incomplete data is only one mandatory feature, possibly qualified algorithms have to suffice. Equally important is their applicability in being executable in parallel. While data parallelism (i.e., SIMD) allows for the execution of the same algorithm multiple times in parallel, operating on different data, parallelizing the algorithms as such (i.e., MIMD), could provide further advantages with respect to the execution runtime. But, special attention has to be paid to the resulting communication complexity. Due to the memory architecture of modern GPUs, i.e., the existence of several memory entities of different sizes, having different access latencies, the risk of resource contention because of synchronizations between algorithms instances is present and needs to be avoided.

This is especially important when aiming at the goal of real-time detection. While modern GPUs are equipped with fast onboard memory, high-speed links to the host system, and massive parallel processors, the identification of real-time behavior of implemented algorithms in specific and the full data acquisition and detection system as such is of interest. In other words, we need to find the optimal trade-off between the level of security (i.e., the observation frequency) and the loss of performance (i.e. forced synchronizations), while at the same time guaranteeing a maximum time-to-detection.

Besides the detection of anomalies and intrusion attempts, the ability of self-protection (and possibly self-healing) is a promising field of research. It must be evaluated what kind of attacks a coprocessor (i.e., the GPU in our case) is vulnerable to. Due to the currently very limited interface between the host system and the device, we expect the number of attacks to be rather small.

Nevertheless, a first step towards self-protection could be the realization of a mutual health check. That is, each of the employed GPU processors (or even GPU threads) randomly checks the status of other processors/threads. If a coprocessor or thread is found to be infiltrated or subverted the tasks assigned to this component must be taken over by the remaining processors/threads. In this case, the infested device must be reseted. To prevent further incidents of the same kind (i.e., to avoid that the attacker uses the same attack vector again), we need a possibility to cure the affected entity from its vulnerability. A first approach towards this ambitious goal would be a thorough and automatic analysis of the attack, followed by a software reconfiguration.

At the very least, a reporting functionality that gives feedback about what led towards the attack should be realized. After all, self-protection, as well as self-healing capabilities must be resource efficient and should be operated as decentralized as possible in order to avoid a single point of failure.

1.5 Document Structure

The present thesis is structured as follows: The related work is presented in Chapter 2. It is divided into three sections and contains related work in the field of modern computer architecture in Section 2.1, coprocessors in general and direct memory access in Section 2.2, and host, as well as network intrusion detection using coprocessors in Section 2.3.

Part II presents the six publications containing the contributions of the present thesis in full length. The first paper, proposing a cost model for tightly coupled asymmetric concurrency, is the subject of Section 3. In Section 4, we present the results of assessing modern GPUs in terms of their ability to be intuitively used as autonomous host auditors, followed by quantified performance degradation figures for various experimental setups, detailed in Section 5. The use of control-flow techniques in a security context, as well as the common weakness of three popular prototypes, is the subject of the paper presented in Section 6. In Section 7, we present the validation of our assumption that by waiving the correspond-

ing programming and runtime frameworks, modern GPUs can be detached from the host in order to function fully autonomously. The fact that computer attacks are non-atomic is taken advantage of in Section 8. Here, we present a probabilistic model that allows for the countering of subversion attempts even before they are been successfully accomplished.

Bibliography

[1] DENNING, P. J. The Working Set Model for Program Behavior. *Communications of the ACM 11*, 5 (May 1968), 323–333. doi:10.1145/363095.363141. 8, 46

[2] FORREST, S., PERELSON, A. S., ALLEN, L., AND CHERUKURI, R. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1st IEEE Symposium on Security and Privacy (S&P 1994)* (Oakland, CA , USA, May 1994), IEEE Computer Society, pp. 202–212. doi:10.1109/RISP.1994.296580. 10, 82, 87

[3] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006. 8, 18, 44, 46, 47, 48, 49

[4] MOLINA, J., AND ARBAUGH, W. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security (ICICS 2002)* (Singapore, Dec. 2002), R. Deng, F. Bao, J. Zhou, and S. Qing, Eds., vol. 2513 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 291–302. doi:10.1007/3-540-36159-6_25. 6, 8, 23, 24, 25, 58

[5] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Computer Society, pp. 421–430. doi:10.1109/ACSAC.2007.21. 10, 43, 53, 82

[6] PASKINS, A. The IEEE 1394 bus. In *Proceedings of New High Capacity Digital Media and Their Applications* (May 1997), pp. 4/1–4/6. (Last checked: August 22, 2012). 9, 68

[7] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[8] RIEDMÜLLER, R., SEEGER, M. M., WOLTHUSEN, S. D., BAIER, H., AND BUSCH, C. Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection. In *Proceedings of the 2nd International Workshop on Security and Communication Networks (IWSCN 2010)* (Karlstad, Sweden, May 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/IWSCN.2010.5497999. 6, 8, 9, 10, 12, 68, 78, 93, 94, 96

[9] SEEGER, M. M. Using Control-Flow Techniques in a Security Context – A Survey on Common Prototypes and their Common Weakness. In *Proceedings of the International Conference on Network Computing and Information Security (NCIS 2011)* (Guilin, China, May 2011), IEEE Computer Society, pp. 133–137. doi:10.1109/NCIS.2011.126. 7, 9, 11, 12

[10] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[11] SEEGER, M. M., AND WOLTHUSEN, S. D. A Model for Partially Asynchronous Observation of Malicious Behavior. In *Proceedings of the 11th Annual Conference Information Security South Africa (ISSA 2012)* (Johannesburg, South Africa, Aug. 2012), IEEE Computer Society. 7, 11, 12

[12] SEEGER, M. M., AND WOLTHUSEN, S. D. Towards Concurrent Data Sampling using GPU Coprocessing. In *Proceedings of the 6th International Workshop on Secure Software Engineering (SecSE 2012)* (Prague, Czech Republic, Aug. 2012), IEEE Computer Society. 6, 10, 12, 28

[13] SEEGER, M. M., WOLTHUSEN, S. D., BUSCH, C., AND BAIER, H. The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation. In *Proceedings of the 9th Annual Conference Information Security South Africa (ISSA 2010)* (Johannesburg, South Africa, Aug. 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/ISSA.2010.5588311. 5, 6, 9, 11, 12, 93, 94, 104

[14] TARAFDAR, A., AND GARG, V. K. Happened Before is the Wrong Model for Potential Causality. Ece-pds-1998-006, University of Texas at Austin, July 1998. (Last checked: August 22, 2012). 7, 102, 106

Chapter 2

Related Work

This chapter provides an overview of related literature in the field of modern computer architecture in Section 2.1, coprocessors and direct memory access in Section 2.2, and host, as well as network intrusion detection using coprocessors in Section 2.3.

While an exhaustive amount of related scientific papers and standard work exists, we focus on contributions most relevant to the present thesis.

2.1 Modern Computer Architecture

The architectural features of modern computer systems are of prime importance to the present thesis. The evolution towards several caching hierarchies, multiple processors, and memory entities supporting different clock speeds is based on a single demand: the demand for faster computer systems. The result of interfering with said high speed memory and processing entities is lower performance.

With system architecture being the heart of any information system, the following section will briefly introduce the history of computer architecture to make clear where today's ubiquitous non-uniform memory architecture comes from.

According to Hellige [31], the term *computer architecture* dates back to the beginning of the 1960s, when scientists started to focus on the earlier less-commonly-regarded aspects of computer systems: technical organization and design structure. The philosophy went away from a construction-only approach towards the organization of the full structure of a computer. Even before the term *system architecture* as such was mentioned in the IBM (international business machines) report RC-160, authored by Johnson in 1959, Speiser, at that time head of the European IBM research laboratories Zurich (Switzerland), authored in 1950 a design guideline in which he described the computer as a "design conflict model", aimed at contrary goals while trying to establish a reasonable equilibrium between hardware and software efforts. Based on these thoughts, in 1961, he published his textbook *Digitale Rechenanlagen* [85]. This book is known to be the first one to elaborate on a hierarchical architecture concept for computer systems.

Today's commodity computers basically follow an architecture which is formally called the *von Neumann architecture* [96]. Besides the fact that Zuse's Z1 already included some of the characteristics of this new architecture (cf. patent application Z23139/GMD Nr. 005/021), it was the technical report entitled *First Draft of a Report on the EDVAC*[1] by von Neumann [96], published in 1945, that outlines what is today known as the von Neumann architecture. Whether or not von Neumann knew about Zuse's work, and if or to what degree he was inspired by the work of Turing (cf. [92]), cannot be answered for sure.

Besides the input/output (IO) unit, the von Neumann architecture is distinguished by a memory unit and a processing unit divided into control and arithmetic logical units (ALU) [96]. In particular, the presence of a memory unit for storing data, as well as programs earned von Neumann the title of being the inventor of the first stored-program computer [25]. Prior to this, computers such as the ENIAC (electronic numerical integrator and computer) had to be programmed by reconfiguring the hardware wiring. This task was known

[1] EDVAC: electronic discrete variable automatic computer

to be very complex and time consuming, described in detail by Goldstine [26], and could only be done "at the cost of much tedious preliminary labor" [27].

According to Flynn's taxonomy [22], "a hierarchical model of computer organizations", the von Neumann architecture falls under the single-instruction single-data (SISD) category. Computers belonging to this category intrinsically suffer from the so-called *von Neumann bottleneck*, a term introduced by Backes [4] in 1978, describing the architectural weakness that occurs when one bus is used to connect the processing unit with the memory unit that holds both data and instructions. As SISD architectures work absolutely sequentially, the bus is the primary limiting factor, as it operates at much lower speed than the CPU. Furthermore, the fact that CPU frequencies rose (and still do, though but at a slower pace) much faster than the speed of buses or access times for memory exacerbates the impact of the von Neumann bottleneck.

Even the gap between clock speeds of modern CPUs and memory access times poses a problem and the von Neumann bottleneck is, therefore, still present. Borkar et al. [6] state in their white paper that, to this extent, "...performance will have to come by other means than boosting the clock speed of large monolithic cores. Instead, the solution is to divide and conquer, breaking up functions into many concurrent operations and distributing these across many small processing units." Today, the effect of memory access times limiting a computer's performance, is known as the *memory wall*, briefly described by Wulf and McKee [102].

Between 1939 and 1944, Aiken, at that time already working at Harvard University, designed and assembled the IBM automatic sequence controlled calculator (ASCC), better known under the name *Harvard Mark I* [17, 40]. In contrast to the von Neumann architecture, the design of the Harvard Mark I was based on using separate memory (i.e., also separate buses) for data and instructions, which was seen as a solution for the root cause of the von Neumann bottleneck (i.e., data and programs stored in the same memory and hence connected by the same bus). As this was a fundamental step forward in the field of computer architecture, this design was given a name: *Harvard architecture*. The Harvard architecture shows a clear improvement over the von Neumann architecture as data and instructions can be loaded in parallel, which is achieved at the cost of complexity. Today, structures based on the Harvard architecture are, due to their performance, applied to data intensive tasks. That is, they are included in the design of digital signal processors (DSP) [28]. While today's commodity computers basically follow the von Neumann architecture, they also incorporate some design features based on the Harvard architecture. For performance considerations, i.e., to counter the von Neumann bottleneck and the memory wall, caches have been introduced to both architectures. In conjunction with prefetching and write-back strategies, the number of accesses on the slower memory entities, as well as synchronization overheads, are tried to be kept to a minimum (cf. [32]).

The increasing number of multiprocessor computers and, with it, the omnipresence of caches, has led to several different memory architectures, of which the best known is the non-uniform memory architecture (NUMA). Much simpler than the NUMA is the uniform memory architecture (UMA), in which the memory access times for each processor are identical, owning to the fact that "all processors have a uniform latency from memory even if the memory is organized into multiple banks" [37]. The main memory, as well as the address space is physically shared between all processors, and allows for each processor to execute the same process. The processor architecture applying UMA is commonly known as symmetrical multiprocessor (SMP) architecture [15]. In contrast to this, NUMA provides each CPU (or each set of CPUs belonging together) with its own local memory and its own local I/O subsystem, while each CPU (or CPU set, respectively) also has access to non-local memory, i.e., the local memory of other CPUs [15]. As different memory access times exist within this memory architecture, it is called non-uniform. Due to the corresponding characteristic properties, UMA is best suited for cases where a single (large) process is executed by many CPUs while NUMA performs best in scenarios where multiple tasks are

executed by different CPUs. Concrete implementations of NUMAs vary not only among system vendors but also between different system series built by the same vendor. The fact that the performance of such architecture depends on its usage (e.g., execution of tailored scientific applications, hosting of large-scale web services, gaming, office, etc.) contributes to this fact. Therefore, NUMAs follow different strategies in terms of, for instance, when to load certain data into a certain cache and when to omit other data. In order to increase the so-called locality of references, different strategies that perform best in different scenarios have been engineered.

In 1991, Ramanathan and Ni [68] provided a thorough analysis of critical factors in NUMA memory management and found out that not replicating data into different caches is, under certain conditions, even better than a full replication. They used "a trace driven simulator written in GNU C++" and state that management policies should be application-dependent, which reflects the above fact that there are various different NUMA implementations. Results on the performance of a certain NUMA management policy always depend on the processor used. To this extent, Zhan and Qin [105] provided, in the same year, performance results measured on a real NUMA multiprocessor[2] and tested for intercommunication overhead and remote access delay. The results are fine-grained and allow the definition of different processing strategies, while – as could be expected – the efficiency of both intercommunication and remote access depends on the task executed. The certainty that there is no perfect strategy fulfilling all requirements is stressed by the fact that even today, research on this very topic is done, as shown in the 2011 paper by Majo and Gross [53], who assess the NUMA performance of an Intel Xeon 5520 (Nehalem) processor. Essentially, they confirm the results from the early days of the NUMA, namely the fact that its performance depends highly on the use case, as well as the way that applications make use of different memory entities.

2.2 Coprocessors and Direct Memory Access

The concept of employing other processing units to assist the CPU has a long history, and today, this concept is realized in information systems of all sizes. In addition to mathematical computations, which constituted a central field of application for coprocessors in the early days, they are now valuable in many different areas.

The following section will introduce different coprocessors and give examples of their corresponding fields of application. As the GPU is of prime importance with respect to the present thesis, we will also detail the physical and logical architecture of this omnipresent and powerful coprocessor, followed by an introduction of the direct memory access concept. At the end of this section, we also present a coprocessor, that is able to operate concurrently and autonomously, but which is unfortunately rather slow, limited in its application scope, and built upon extraordinary hardware.

Coprocessors are special-purpose processors that assist the CPU by relieving it from dedicated tasks. A coprocessor usually has a limited instruction set and is designed to efficiently perform a certain task at much higher speed than the CPU could do it. An overview of the requirements of a secure coprocessor, its architecture, and the threats that need to be countered are summarized by Smith and Weingart [80]. Earlier works by Smith cover, for instance, the application of secure coprocessors and research issues [78], and propose a balance between security and ease of programmability [79]. Furthermore, Yee has in detail presented the security properties of secure coprocessors available before 1994, and has also demonstrated "how physical security requirements may be isolated to the secure coprocessor" in [103]. How coprocessors fulfilling corresponding properties can be used in electronic commerce applications is also discussed by Yee in [104]. All these early works have the fact in common that the proposed architectures and prototypes are

[2] They used the BBN GP1000 (cf. [2]).

capable of performing limited cryptographic computations or are used to securely store cryptographic keys.

From a historical point of view, von Neumann introduced the first coprocessor in his report on the EDVAC [96], where he mentioned not only the CPU but also an assistive processing unit in charge of arithmetic and logical operations (ALU: arithmetic logic unit). As computer architecture matured over time, former coprocessors such as the ALU or later the FPU (floating point unit) became integral parts of the CPU [60], which saved silicon and reduced latencies caused by slow buses [33]. Besides its application in digital signal processing, today, the field of graphics computations is well-known for the application of powerful coprocessors. This is not only because of the rising demand for high resolution images, but is also due to the fact that modern graphics processing units (GPU) are suitable for non-graphics computations. To this extent, the term *general purpose computation on graphics processing units* (GPGPU) mainly refers today to the use of modern GPUs together with vendor dependent programming and runtime frameworks such as the CUDA (compute unified device architecture) SDK (software development kit) from NVIDIA, or the Stream SDK from AMD (advanced micro devices) and ATI (ATI technologies inc.), respectively. These coprocessors draw their massive power mainly from their parallel architecture, which allows the execution of single or multiple instructions on multiple data. According to Flynn's taxonomy [22], these architectures are called single instruction, multiple data (SIMD), and multiple instructions, multiple data (MIMD). In addition to the architecture, it is the possession of several local off-chip (i.e., local, constant, texture and global memory) and on-chip (i.e., registers, caches and shared memory) memory entities of different sizes and access times that makes a GPU fast. This gives application programmers the chance to take advantage of bulk data transfers between host and coprocessor, hence keeping the need for time-intensive interactions over the interconnection bus to a minimum. Off-chip memory is just another term for the GPU's device DRAM (dynamic random access memory) memory, which serves as an interface between the GPU as such and its connected host.

In order to provide an insight into the architecture of a modern GPU, we now detail the physical memory and the logical execution architecture of a Gainward GeForce GTC 260 GS graphics card. Facts and figures are based on the NVIDIA CUDA programming guide (cf. [64]). The device memory is divided into three regions. Usually, the largest memory available is the global memory. It is uncached and thus generally the slowest memory. An alternative path to regions of the global memory is available by making use of the texture memory. It also resides within the device memory, but, in contrast to the global memory, the texture memory has a 6 to 8 KB on-chip cache per multiprocessor. The smallest region of device memory is called constant memory. While it is only 64 KB large, it is referenced to an 8 KB on-chip constant cache. As its small size and the availability of a cache already suggest, the constant memory is very fast. That is, as long as concerned threats read from the same address, accessing the constant cache is as fast as accessing a register. On-chip memory is generally the smallest and fastest memory available. Each streaming processor has access to a set of registers[3]. With respect to memory speed, the registers are the fastest, while with 32 bit, they are also the smallest memory available. As one register belongs to one streaming processor at a time only, a shared memory of 16 KB, organized in 16 banks, connects all streaming processors belonging to the same multiprocessor. The shared memory is as fast as the registers, as long as no bank conflicts occur. In addition to private registers, each streaming processor has access to local memory. It is worth mentioning that local memory is physically a part of the global memory and thus off-chip. Since global memory is slow and uncached, the same holds for local memory. The programming framework of the NVIDIA CUDA technology allows for serial programming, while the execution model provides a massive parallel environment. A function written in the programming language CUDA C, tagged for device execution, is called a kernel. This kernel is executed by several

[3] 16.384 per streaming multiprocessor in the case of a modern Gainward GeForce GTC 260 GS.

blocks. While each block contains the actual threads, the blocks themselves are grouped into one grid. The fact that only one kernel at a time can be processed implies two things: First, kernels are processed sequentially, which implies synchronization overhead. Second, as only one grid is executed at a time, its blocks are distributed over all available streaming multiprocessors. From the communication's point of view, threads within the same block may synchronize via shared memory. As shared memory is a per-multiprocessor memory, one block cannot spread over two or more multiprocessors. This also implies that threads of different blocks cannot communicate intuitively[4]. That is, they operate in total independence. Ideally, all threads within one warp (a logical unit meaning a group of 32 parallel threads) agree upon the same instruction path, as this allows the full warp to finish with maximum speed. The model behind this is called SIMT (single instruction, multiple threads).

Graphics computations can be complex, and the amount of data subject to these computations may be vast. In order to provide a high speed connection with the host, the GPU is attached to the northbridge, i.e., the same chip CPU and RAM are directly connected to, and uses the PCIe (peripheral component interconnect express) bus for intercommunication. In contrast to this, USB, FireWire, and even other PCIe interfaces are attached to the southbridge and, therefore, suffer from high latency when, for instance, communicating with the host's CPU or system memory. One common way of lowering latency, and, in particular, the impact data transfers can have on the performance of the CPU (and thus, the whole system), is through the use of the direct memory access (DMA) technology.

"*Direct Memory Access (DMA)* is a means by which devices can exchange data with memory or with each other, without requiring intervention by the processor. Standard DMA allows a device to exchange data with memory, but not with another device. *Bus Mastering DMA* allows two devices to communicate directly with each other. The advantage of using DMA is that it reduces the load on the processor, allowing it to perform other tasks." [88, p.21]

Bus mastering is also called *third-party DMA*, which refers to the fact that the device itself (e.g., GPU) provides a DMA controller, and thus allows data to be transferred without passing through the CPU [88].

In addition to intrusion detection (cf. Section 2.3), coprocessors are applied in other IT security related fields. Two prominent examples are cryptography and digital forensics. Cryptographic coprocessors have a variety of physical manifestations with smart cards being the most prominent one. Smart cards with dedicated cryptographic coprocessors are often used in conjunction with public-key infrastructures. That is, a user's private key, which is used to establish a secure communication environment, is stored on a smart card [29]. The advantage of a smart card, in contrast to other coprocessors, is the convenience of easily carrying one at all times, while their application often does not require physical contact with the smart card reader, as cards with a contactless data transmission feature exist. Already in 1994, Clark and Hoffman [12] envisioned a smart card's applicability beyond pure access control and proposed to have computers boot from smart cards in order to ensure boot sequence integrity, which, for that time, can be seen as a modern way of trusted computing.

In terms of execution speed and data throughput, a smart card is much slower than, for instance, a desktop computer or dedicated hardware components. A typical AES (advanced encryption standard) computation with a 128 bit key and a 16 byte block takes 20 ms (6.4 kbit/s) on a smart card with a 16 bit CPU running at 4.9 MHz, opposed to 160 ns (400 Mbit/s) on a computer running a Pentium 4 with a 3.5 GHz CPU and 1.9 ns (34 Gbit/s) on an AES hardware component [69, p.141]. Such hardware components are generally integrated into desktop computers and servers, and are interconnected through the PCI (or PCIe) bus. They can be seen as autonomous, as they are basically a full computer

[4]The only way to synchronize blocks is to make use of global memory. That is, after one kernel is finished, another one is started, using the results of the former, which are stored in global memory.

on a PCI-card. For instance, the IBM PCIe cryptographic coprocessor contains a "CPU, encryption hardware, RAM, persistent memory, hardware random number generator, time of day clock, infrastructure firmware, and software" [39]. Due to their equipment, such coprocessors are of course much more sophisticated than smart cards, and are capable of executing a wide range of algorithms including SHA 512 and RSA, for example (cf. [39]).

Field programmable gate arrays (FPGA) are yet another example of powerful coprocessors that are also applicable in cryptographic tasks. As is implied by its name, an FPGA offers a programming interface that allows the implementation of custom logics and algorithms. Ease of configuration is the main advantage held by an FPGA over an application-specific integrated circuit (ASIC), which is designed for a specific task. An ASIC can execute this specific task very quickly, but in turn, cannot be reconfigured easily.

In the field of digital forensics, coprocessors are mainly used for data acquisition. While perpetuating digital evidence stored on a computer's hard disk drive is straightforward, additional hardware is needed in order to capture data from volatile memory, such as the main memory, in a sound way, so that it may be used as evidence in court. One of the most critical functional requirements for said hardware is that it must be capable of creating an exact copy of the data in question without altering the system state. Various types of coprocessors are applied in order to achieve this. Without purchasing special-purpose hardware, FireWire technology can be used to connect one computer with another and in order to gain access to the target system without the target system becoming aware of the connection [19].

Both cryptography and digital forensics are, up to a certain extent, fields in which the use of GPUs is applicable. In 2005, Cook et al. [13] assessed the feasibility of moderns GPUs for use in cryptographic processing, and they conclude that, given the available APIs, GPUs are not suited to process symmetric key ciphers, though they may be used for encryption and decryption of, for instance, graphical images, in order to avoid leaving a temporal footprint of plain text in system memory.

Two years later, Harrison and Waldron [30] presented results from their implementation and testing of a GPU version of the AES block cipher encryption algorithm. Similarly to Cook et al., the authors report a very high CPU utilization when the GPU program is running, and benchmark comparisons of existing CPU implementations of the AES algorithm revealed no performance advantage on the side of the GPU. One reason for this is the fact that GPUs benefit mostly from working on large bulk data and minimum host interaction. A technically detailed description of a GPU-based AES algorithm including performance results is also given in the master thesis by Rosenberg [71].

In 2008, Korobitsin and Ilyin [45] presented a GPU implementation of the GOST-28147 (cf. [18]) algorithm for symmetric ciphering, and reported a performance increase between two and three times for their specific test setup. Based on this, the authors conclude that "the GPU can (be) used to offload ciphering-related tasks from the CPU at times when the GPU stays idle", despite the fact that no performance improvement could be measured in the GPU-based AES implementation.

The validation that modern GPUs can be used within the field of digital forensics was provided by Marziale et al. [54] in 2007. The authors presented the results of GPU involvement in the task of file carving (i.e., finding files by a set of predefined header and footer definitions). In all test scenarios, which differed in the number of definitions to search for, as well as amount of data to search in, the GPU performed best.

Petroni et al. [66] presented a coprocessor-based kernel runtime integrity monitor (CoPilot) in 2004. This prototype is basically a single-board computer on a PCI add-in card, and is able to establish a link between two computers. While initially presented as an off-host host observer, it can also function as a forensics coprocessor for the task of data acquisition. In the same year, Carrier and Grand [7] proposed a hardware-based memory acquisition procedure. The authors described an approach based on a hardware expansion card, which is installed into a PCI bus slot before an incident occurs, thus allowing its application when

needed. A method of preparing a computer system for the acquisition of volatile data is also proposed by Libster and Kornblum [50], who suggest the integration of a memory acquisition mechanism into the operating system, thus, rendering the necessity for externalities obsolete.

2.3 Intrusion Detection using Coprocessors

Works related to intrusion detection using coprocessors are divided into three parts. In Section 2.3.1, literature directly concerned with host intrusion detection using coprocessors is discussed, while in Section 2.3.2, related work within the focus of network intrusion detection using coprocessors is presented. Finally, in Section 2.3.3, literature that provides no direct coprocessor-based approaches, but nevertheless contributes to the broad field of using off-host components to secure the host as such, is briefly presented.

2.3.1 Host Intrusion Detection using Coprocessors

While the concept of using coprocessors for special tasks has a long history, reaching back to the early days of computers (cf. von Neumann [96]), their application for purposes beyond their intended scope is a rather young tradition. Considering the fact that a coprocessor is a special-purpose processor uniquely designed to fulfill a dedicated task at maximum speed, this is no surprise. To the best of our knowledge, Molina [57] was the first to propose the use of coprocessors for host intrusion detection in his master thesis in 2001. Based on his thesis, Molina and Arbaugh [58] published the idea in December 2002. Even though Zhang et al. [106] had already published their paper proposing secure coprocessor-based intrusion detection in July 2002, this work includes a reference to an early and unpublished version of the December 2002 paper from Molina and Arbaugh[5].

Molina and Arbaugh present a definition of what they call an *independent auditor*, and detail a specific implementation using an embedded system attached to a PCI bus. In [58], the major advantage of an observation and auditing mechanism that does not rely on the integrity of the host operating system is already explained: the auditor and, in particular, its results are trustworthy even if the concerned host operating system has been compromised. In cases where the auditor is installed on the machine to be audited, an attacker may be able to subvert the auditor itself. For a proof of concept, Molina and Arbaugh used an evaluation board from Intel called EBSA-285, which is based on the Intel StrongARM SA-110 processor, and installed the open source advanced intrusion detection environment (AIDE) (cf. [48]), in order to perform file system integrity checks. The performance impact on the target system was reported to be 5% in the worst case, and the average transfer rate was 1.4 Mbit/s.

The approach presented by Molina and Arbaugh highlights the problems of auditors that rely on host system integrity. Their proof of concept provides validation of the assumption that coprocessors can be used to perform host intrusion detection. Due to the fact that they establish a direct connection between their coprocessor and the host's hard disk drive, the performance degradation remains rather low at the cost of transmission speed. Therefore, it remains questionable whether using an external, low-speed CPU connected to a high latency memory entity can provide a reasonable level of security. Despite the doubful applicability in real-life scenarios, the defined properties an auditor must fulfill in order to be called *independent* are an important contribution. Even though none of the works from Smith et al. were referenced, leaving aside cryptographic qualifications, said properties show a good congruence with the requirements of a secure coprocessor published by Smith and Weingart [80] in 1999.

As already mentioned, Zhang et al. [106] proposed the application of a secure coprocessor for host intrusion detection in July 2002. The presented approach is closely related

[5] Reference [4] in [106].

to that of Molina and Arbaugh, as it uses an external auxiliary coprocessor to perform integrity checks on the host. In contrast to Molina and Arbaugh, Zhang et al. used the IBM 4758 PCI cryptographic coprocessor. It may be interesting to know that this coprocessor was among others designed by Smith and Weingart. Zhang et al. summarize the advantages of a secure coprocessor in four points: the independence from the host's OS, the narrow interface between host and coprocessor, the ability to boot securely, and its ability to act as a trusted observer. The authors argue that the fact that host vulnerabilities do not affect the proper functioning of the coprocessor can be regarded as self-protection. In contrast to this rather passive self-protection feature, McEvoy and Wolthusen [56] previously introduced a mechanism, distinguished by distributed computations, that is difficult to subvert, even with full knowledge of the mechanism, which can be seen as active self-protection. Zhang et al. name three different scenarios in which coprocessors can be used to secure the host: checking invariants that belong to the data structure of the host OS kernel, file and file system integrity checking, and virus detection. With respect to the observation of kernel invariants, the work from Zhang et al. differs from existing on-host approaches (cf. e.g., [23]) in its degree of abstraction. With respect to virus detection capabilities, the authors do only state that their approach is "much less intrusive than (using) a tool like Norton Utilities". They also say that, due to the fact that their technique relies on sampling (which is rather slow, as the PCI bus is used), attacks may not be detected, and thus only the likelihood of a successful subversion attempt can be reduced. While there is no absolute security, any intrusion detection approach *only* reduces the likelihood. Nevertheless, sampling speed does play a vital role in increasing the chance of detecting attacks while they are ongoing.

Based on the achievements of 2002 (cf. [58]), in 2004, Petroni et al. [66] proposed a prototype of a coprocessor-based kernel integrity monitor called Copilot. Still based on the EBSA-285 evaluation board, Copilot focuses on the physical memory of the host rather than its hard disk drive. To be precise, it is equipped with knowledge about how an uncompromised system looks, and compares its observations with this knowledge. To this extent, Copilot does not aim to protect systems from being infected, but to prevent the installation of rootkits on those systems that have already been infected. To achieve this goal, it simply compares *known good hashes* of memory that contains kernel or LKM (loadable kernel module) text, as well as memory that contains jump tables of kernel function pointers, with frequently recalculated hashes of the same data. Petroni et al. note that Copilot was able to detect the presence of 12 real-world rootkits, and they give further insight into how the performance degradation was measured. That is, they used the STREAM [55] and WebStone [89] benchmark and compared the data throughput with and without Copilot being on duty. As a result, they report a performance degradation as low as 0.84% on the host being observed when the integrity checks are repeated every 30 seconds.

The Copilot prototype is an impressive example regarding the feasibility of a secure coprocessor performing host integrity validation. Besides the limitations discussed by Petroni et al. themselves, some questions regarding the reported performance degradation remain open. For instance, it remains unclear how fast the coprocessor could transfer data off the host, as well as how much data was copied during each run. This information would give the performance penalties more significance. It is also important to keep in mind that the figures reported in [66] reflect the hashing of only one kernel jump table, which is fine for a proof of concept, but not enough for the real-life usability of Copilot.

As a common weakness, all proposed approaches make use of external auxiliary coprocessors, which is counterproductive with regard to their applicability in private households. In detail, Molina [57], Molina and Arbaugh [58], and Petroni et al. [66] used the EBSA-285 evaluation board, and Zhang et al. [106] applied the IBM 4758 PCI cryptographic coprocessor.

In 2010, Fechner [21] presented a hash-based approach that proposes the use of a GPU for SHA-1 (secure hash algorithm 1) calculations in order to detect virus signatures in host

data. To this extent, a host process copies the data in question off the host, onto the GPU memory, where the actual assessment is done, taking advantage of the GPU's architecture in terms of executing a single command on multiple data (i.e., SIMD), which does lead to a performance improvement compared to a host-only approach. The fact that GPUs are beneficial for such tasks is not new and has been presented by other researchers before, especially in the context of network intrusion detection (cf. Section 2.3.2). With respect to the author's aim to provide additional merit concerning host security, the presented approach seems to be inappropriate for at least one major reason. That is, the ability of the GPU to perform its tasks depends highly upon the host components transferring the data to audit onto the GPU. While attacking host-side services can be achieved by even moderately skilled adversaries, i.e., shutting them down or invalidating the data to be assessed, a combination of GPU and CPU services does not provide additional value in terms of host security.

2.3.2 Network Intrusion Detection using Coprocessors

The ease of accessing network data, in contrast to the efforts one must undertake in order to access data stored on a host system (e.g., desktop computer or laptop), may be the main reason why coprocessors are more prominent in the field of network intrusion detection than they are in the field of host intrusion detection. While one of the advantages of using coprocessors for host observation is the increase in security, due to a set of properties they have to fulfill (cf. e.g., [58, 80]), the reason to apply coprocessors for auditing network data is solely founded by the increase in speed. As network bandwidths have come a long way from several kilobits per second, in the early days of the Internet [72], to several hundred gigabits per second today [62], using dedicated, high-speed devices for deep packet inspection (i.e., pattern matching on the payload of network packets) is necessary in order to keep latency to a minimum.

Using GPUs for host intrusion detection purposes is a rather new discipline. Therefore, this section also aims to provide references for concepts that have led to successes in the adjacent field of network intrusion detection using coprocessors.

There are basically two types of coprocessors applied in practice: Field programmable gate arrays (FPGA) and graphics processing units (GPU).

FPGAs for Network Intrusion Detection

In addition to early works in the field of text processing using FPGAs (cf. e.g., [51, 67]), Sidhu et al. [76] are recognized for presenting the first FPGA implementation of a well-known string matching algorithm (i.e., the Knuth-Morris-Pratt (KMP) algorithm for fast pattern matching in strings (cf. [44])) in 1999. While their main focus was on the reduction of mapping time, which "refers to the time (needed) to compile, place and route the logic to be used on the FPGA", by using self-reconfiguration on multicontext FGPAs, the authors implemented the KMP algorithm in order to emphasize the runtime advantage of their approach. While the mapping process is time consuming and must be repeated for every problem instance using CAD (computer aided design) tools, having the FPGA itself generate future configurations based on previous ones (i.e., self-configuration) has an enormous runtime advantage. Searching for an 8 character long string in a text of 10^4 characters, Sidhu et al. reported a speedup in mapping of about $6 \cdot 10^6$, compared to using a CAD tool, and an improvement of 16.6 over a software version (i.e., executed on a host) of the KMP algorithm.

Even though Sidhu et al. include FPGA implementation details of a pattern matching algorithm, as well as results of an initial performance analysis, their focal point of the work is on self-configuration of the FPGA. Two years later, Sidhu and Prasanne [75] presented a realization of a nondeterministic finite automaton (NFA) running on an FPGA. This novel implementation is compared to a deterministic finite automaton (DFA) executed on a serial

machine. The performance comparison reflects the space requirement, the construction time, and the time per text character. With respect to the first property, the NFA reports quadratic space requirement, whereas the DFA experiences exponential growth. In the worst reported case, the construction time of the DFA took more than 24 hours, as opposed to less than a tenth of a second in all NFA cases. Concerning the time needed to process each character, the DFA version reports a constant period of 30.88 seconds, whereas the time needed by the NFA increases roughly linearly. Nevertheless, "assuming the linear increase holds, the time required by the FPGA (i.e., NFA) to process a text character will be lower than the time required by software (i.e., DFA) even for regular expressions hundreds of characters in length".

While not directly related to information security, the previously stated early applications of FPGAs for pattern matching give an idea about the the complexity in terms of (re-) reconfiguring said devices. This also holds for the first security focused FPGA-approaches, as the following examples confirm.

To the best of our knowledge, it was Hutchings et al. [38], in their paper from April 2002, who first proposed the use of coprocessors for network intrusion detection. Their FPGA-based regular-expression module generator eventually generates an EDIF (electronic design interchange format) netlist, which is, in turn, used to create an FPGA bitstream. In order to investigate the performance of their approach, the authors present a comparison to a software implementation of the GNU regex program. To validate its applicability when combined with network intrusion detection systems (NIDS), the open source NIDS Snort[6] [70] was selected. The tests revealed that the software implementation performed better for smaller regular expressions (i.e., ≤ 47 non-meta characters), but was outperformed by the FPGA implementation, which was based on the NFA approach proposed by [75], for larger expression in terms of latency, throughput, and CPU utilization by more than 600 times, in the best case. Based on this result, the authors conclude that FPGA-based string matching can reduce CPU workload while at the same time processing more data packets in less time.

Until today, research in the field of utilizing FPGAs, specifically, and coprocessors in general, for host intrusion detection (i.e., DPI) is mostly concerned with proposing new architectures and algorithms implemented in the latest hardware in order to consume less memory and to assess more packets per second. To this extent, we will now continue to briefly present remarkable contributions, mostly in chronological order:

In September 2002, Cho et al. [10] presented an FPGA-assisted software firewall, capable of processing 2.88 Gbit/s of traffic, in contrast to an original maximum of about 60 Mbit/s. Their test environment used the Hogwash[7] [3, p.402] filter, which is wrapped around the Snort IDS. Besides the actual performed measurements, the authors hypothesize that a properly tuned version of their approach could allow the assessment of 6 Gbit/s without further latency.

One year later, Dharmapurikar et al. [16] proposed the use of parallel bloom filters for string matching on FPGAs. While the data processing speed was reported to be 2.4 Gbit/s, this performance was measured while scanning a set of 10,000 strings. In addition to the advantage over existing approaches, that is, the ability to handle larger data by consuming reasonable resources, the authors also state that applying bloom filters has the drawback of causing false positive alarms, but never false negatives. Dharmapurikar et al. use a second process to take care of said false alarms.

In the same year, Sourdis and Pnevmatikatos [82] achieved a throughput of 11 Gbit/s. In contrast to previous work, the authors employ neither finite automata nor packet-level parallelism (cf. [59]). Instead, full-width comparators are used. They state that by easily extending said comparators (i.e., adding more resources), even higher throughputs can be achieved.

[6] http://www.snort.org
[7] http://hogwash.sourceforge.net

Baker and Prasanna [5], Cho and Mangione-Smith [8], and Clark and Schimmel [11] presented their contributions concerning the size of the FPGA logic at the 12^{th} annual IEEE symposium on field-programmable custom computing machines (FCCM 2004). Baker and Prasanna focus on a pre-filtering architecture that reduces the need for duplication within comparators. While increasing the area-time performance, this technique also increases the number of false positive alarms. Cho and Mangione-Smith set their focus on reusing parts of the logic filter in order to reduce the actual filter size. Additionally, a simple rearranging schema for better memory utilization is proposed. Both methods combined make it possible to place 230,300 gates onto a single Spartan 3 FPGA, while allowing auditing network traffic at 3.2 Gbit/s. Clark and Schimmel presented a design for a multi-character decoder NFA technique, which theoretically allows pattern matching to be performed at network rates of 100 Gbit/s and more. The major advantage over existing approaches (next to the assumption of an input width of 512 bits) is the fact that their technique allows the development of circuits that are capable of processing more than one character per clock cycle. The authors also compared their decoder NFA approach to experimental results from other authors (e.g., [82]) that revealed a much lower performance at real input widths (i.e., 32 bit).

As an extract of his master thesis (cf. [81]), in 2005, Sourdis et al. [83] presented an evaluation of different FPGA-based pattern matching techniques, which also revealed that the approach from Clark and Schimmel may be promising for the future, but does show a reduction in performance for common input widths.

An architecture that can be implemented for ASICs, as well as for FPGAs was presented by Cho and Mangione-Smith [9] in the same year. The authors make use of a pattern detection module and a long pattern state machine in order to deal with patterns of different length efficiently. Another remarkable contribution is the fact that the final filter can be adjusted without the burden of reconfiguring the hardware as such.

In 2008, Sourdis et al. [84] detailed a comparison of two competing approaches, i.e., decoded partial content-addressable memory (DpCAM) and perfect hashing memory (PHmem), in terms of performance and cost per area. In contrast to PHmem that requires both memory and logic, DpCAM is based on logic only. Nevertheless, the authors conclude that the best results are achieved using a combination of the two.

As part of their adaptive threat prevention system (ATPS) (cf. [42]), proposed in 2007, Kim et al. [43] presented a multi-hash mechanism in 2009 based on a novel filtering technique called table-driven bottom-up tree for exact string matching. The bottom-up technique maximizes memory efficiency while the application of multiple hash tables improves the run time performance. The authors report a throughput of *only* 2 Gbit/s, together with a note that up to 10 Gbit/s may be possible by optimizing the clock speed. Here, it is worth mentioning that the performance does not suffer from the higher number of signatures and, in particular, not from patterns of variable size (i.e., usage of wildcards such as + or *), which is a valuable feature.

In recent years, research interest regarding the use of FPGAs in network intrusion detection has declined, and GPUs are now within the focal point of industry and academia. The fact that they can be regarded as ubiquitous (i.e., commodity) coprocessors allows for their application in professional, as well as private use, and the existence of programming and runtime frameworks for different platforms (e.g., Windows, Linux and Mac OS) attracts a large community.

As already mentioned, applying FPGAs for network intrusion detection has a strong focus on performance in terms of processing as much packets as possible per time unit. While, of course, performance figures are important for any detection mechanism, performing host intrusion detection according to the busy-wait approach (i.e., at maximum speed) causes a considerable degree of performance degradation due to forced synchronizations between the host's memory entities [73].

In the following section, we will present remarkable publications concerning the field of network intrusion detection using modern GPUs.

GPUs for Network Intrusion Detection

To the best of our knowledge, Jacob and Brodley [41] were the first to propose the use of GPUs for network intrusion detection in 2006. Their prototype, called *PixelSnort*, included a rewrite of the string matching module of the traditional Snort and achieved a performance boost of up to 40% at a line speed of 100 Mbit/s. Due to the high performance of the applied GPU (i.e., 54 GFlops) and GPUs in general compared to CPUs (i.e., 5.6 GFlops on an Intel Pentium 4 Xeon 2,800 MHz), as well as their parallel architecture, GPUs are well suited to perform parallelizable tasks such as string matching. The authors used a programming language called Cg (C for graphics) to implement the needed functionality as fragment shaders. That is, instead of just copying Snort's Aho-Corasick algorithm (cf. [1]), they took advantage of the parallel rendering pipelines of the GPU and created a new parallel-packet string matching algorithm based on the Knuth-Morris-Pratt (cf. [44]) algorithm. While reporting a performance advantage between 50% and 85% in terms of CPU and memory intensive tasks, PixelSnort does not yield measurable benefits for file system intensive tasks.

In 2008, Vasiliadis et al. [93] presented a prototype for using the GPU as a pattern matching coprocessor in a network intrusion detection system (i.e., Snort), which resembles that of Jacob and Brodley somewhat. In contrast to their approach, the prototype, named Gnort, by Vasiliadis et al. was implemented using CUDA (compute unified device architecture) technology (cf. [63]), a proprietary programming and runtime framework developed by NVIDIA, published in June 2007. In their paper, the authors detail that using single pattern matching algorithms (e.g., Knuth-Morris-Pratt) does not perform well on a parallel architecture such as the one offered by a GPU. In a test scenario where the UDP (user datagram protocol) packet size was set to 1,500 bytes, results from a GPU implementation of the multi pattern matching algorithm proposed by Aho and Corasick using an NVIDIA GeForce 8600GT card revealed a performance increase of 3.2 (at a line speed of 2.3 Gbit/s) compared to the CPU version.

In contrast to the proof of concept presented in [74], the Jacob and Brodley [41] apply the high-level shader language Cg and Vasiliadis et al. [93] make use of the CUDA framework. Both approach are suitable for scanning network traffic but pose serious risks for tasks concerning host security.

A non-security related performance comparison concerning the runtime of the naïve, Knuth-Morris-Pratt, Boyer-Moore-Horspool (cf. [34]), and Quick-Search (cf. [86]) string matching algorithms implemented on a GPU using CUDA was presented in 2009 by Kouzinopoulos and Margaritis [46]. The authors report a performance increase for all four algorithms ranging from 1.5 to 24, depending on the algorithm and the length of the search string. Furthermore, they reveal that incorporating the internal memory architecture of a GPU (i.e., using shared instead of global memory) provides extreme speedups for all algorithms besides the naïve one; here, the speedup was *only* 2.5.

In 2008, Huang et al. [36] proposed a GPU-based multi-pattern matching algorithm derived from the idea of Wu-Manber (cf. [101]) and reported a doubling in execution performance, compared to the Wu-Manber algorithm used in Snort. For their test scenarios, the authors applied the publicly available network data captured during the DefCon[8] capture the flag contest in 2001 (cf. [87]). Regarding the performance assessment, five different configurations were tested that differed in terms of how the data in question was fed into and stored on the GPU. In any case, the GPU implementation outperformed the sequential CPU version. As mentioned previously, the CUDA framework was published in June 2007. Nevertheless, the authors chose to apply open graphics library (GL) APIs to control the communication between CPU and GPU.

In 2009, Wu et al. [100] presented a signature matching model that focuses on pattern and search text partitioning in order to balance the loads between the corresponding multi

[8] http://www.defcon.org

processors of a GPU. They refer to their technique as balanced pattern set partition method (BPSPM), which achieves a better performance because of two features: Different multiprocessors search for different pattern subsets in the same input text while different stream processors inside a multiprocessor search for the same pattern but in different parts of the input text. Compared to the performance of the naïve way of using the GPU for string matching, the BPSPM always succeeded.

While the overview over the research concerning GPU-accelerated pattern matching for applications in computer security presented by Gee [24] in 2009 is not exhaustive, in the appendix, the source code of a CUDA version of a bloom filter is presented.

In 2010, Tumeo et al. presented the performance results of a GPU implementation of the Aho-Corasick pattern matching algorithm. With optimizations done in terms of the way the data is loaded into the GPU and the way the transition matrix is consumed by the algorithm, the authors report a throughput of 6.7 Gbit/s using a NVIDIA Tesla C1060 card. One year later, Tumeo et al. [91] compared the results from 2010 to results using the same algorithm on the latest hardware from NVIDIA (i.e., Fermi Architecture), which showed a maximum throughput of 22.7 Gbit/s in the best case. The fact that GPUs can also be used by malware writers in order to obfuscate their code was stressed by Vasiliadis et al. [94] in 2010. The authors present how unpacking and run-time polymorphism can benefit from modern GPUs. For instance, by taking advantage of the high computational power of a GPU, malware authors can apply complex packing algorithms whose execution on the CPU "would take a prohibitively long time to complete". Furthermore, since GPU code is not based on the well-known IA-32 (Intel architecture 32 bit) instruction set architecture (ISA), existing detection mechanism may not even be able to assess such code. Vasiliadis et al. aim to raise awareness concerning the application of GPUs for subversive tasks and provide details about proof of concept implementations of the proposed concealing techniques.

At the beginning of 2011, a technical, thoroughly-described implementation of a GPU-based multi-session deep packed inspection mechanism, presented by Huang et al. [35] in 2010, appeared in the collection *Intrusion Detection Systems*, edited by Skrobanek [77]. The authors used the Snort signatures and the same network data they have used in previous work (cf. [36]) and present all parts of their prototype (i.e., data flow, data structure and control-flow mechanism) in detail. While in terms of performance, the authors state that their approach "is potentially as good as other FPGA or ASIC solutions", they also emphasize the fact that their method is software-based, and can thus be reconfigured easily and is able to operate on commodity hardware, found in modern desktop computers and even laptops.

Very recently, Vasiliadis et al. [95] presented an implementation of a GPU-based pattern matching library, which can be used for a variety of scenarios, including real-time network intrusion detection. Additionally, a buffering scheme for transferring network data onto the GPU is proposed, which partly eliminates the high overhead caused by the PCIe bus. While, without any buffering, the transfer speed between host and GPU amounts to 8 Gbit/s, the buffering schema allows for a rate of over 45 Gbit/s. The performance analysis of the string matching mechanism including said schema revealed a maximum throughput of 29.7 Gbit/s using a NVIDIA GeForce GTX480 card.

The previous references verify the massive computational power of modern GPUs and make further research towards their application within the field of host security promising.

In the following section, we will present remarkable publications showing indirect connections to the present thesis.

2.3.3 Related Approaches

In 2005, Williams [98, 97] presented a coprocessing intrusion detection system called CuPIDS which takes advantage of the characteristics of a symmetric multiprocessing (SMP)

system[9]. That is, a dedicated processor is used to execute a so-called shadow process, observing an application running on a different processor within the same computer. The authors state that their prototype was able to guarantee real-time (cf. [47]) reaction and responses to attacks targeted at the process under observation, causing about 15% performance degradation under certain circumstances. Compared to existing uni-processor-based approaches such as the IDIOT intrusion detection system (cf. [14]), this is a major advantage. CuPIDS is able to perform three types of protective activities, being "application startup/shutdown validation, state monitoring, including invariant testing, and execution monitoring".

In contrast to the present work, Williams mainly focused on the detection performance of their approach, paying almost no attention to the security of the security mechanism as such. Furthermore, using a dedicated CPU core of an SMP system does indeed have advantages over traditional solutions, but cannot be seen as a coprocessor approach as, for instance, no exclusive memory exists, which allows attackers to directly tamper the observer's resources.

Nightingale et al. [61] also propose the use of multi-core systems to accelerate run time security checks. The authors parallelize security checks that would otherwise be executed in sequence. That is, a process in question is replayed to a dedicated core. While the original keeps on executing using speculative execution, one of three possible security checks is executed on the copy: sensitive data analysis, on-access virus scanning, or taint propagation. Tests using an 8-core system have revealed speedups between 2 (taint propagation check) and 7.5 (sensitive data analysis).

Similarly to the work of Williams, Nightingale et al. do not consider protection of the monitoring system itself and rather focus on acceleration of security checks using the processing power of multi-core CPUs.

Besides the subversion of services operating on the host and attacks against the intrusion detection mechanism itself, peripherals' firmware can also pose a security risk. For instance, in 2010, Loïc et al. [20] presented a proof of concept for attacking a network card's firmware. Just recently, Li et al. [49] proposed "software-only attestation protocols to verify the integrity of peripherals' firmware". A prototypical implementation called VIPER was able to detect all known software-based attacks. VIPER is based on a challenge-response protocol between the host's CPU and the peripherals. The verifier program inside VIPER has benign copies of all firmwares concerning the computer system it operates on. Additionally, a checksum simulator is present, generating the challenges and the expected responses from the corresponding firmware. On the device side, a checksum function tuned for creating minimal overhead is implemented. In addition to a comparison of the expected and the actual checksum returned by each peripheral, the time the checksum calculation takes is also assessed, as large deviations may be caused by malicious activities.

The authors evaluate their approach in terms of security by implementing different kinds of attacks (e.g., Ethernet-based proxy attack). All attack scenarios tested can be detected due to causing the checksum computation to consume notably more time. In addition, a discussion covering problems, limitations, and known issues is detailed, mainly focusing on practical problems concerning the feasibility of the proposed approach and does not regard its security. While VIPER is able to validate the integrity of selected peripherals' firmware, we believe that a multi-staged attack could simply shutdown VIPER and proceed to the firmware of interest afterwards. Therefore, combining this approach with off-host host intrusion detection mechanisms seems promising.

The general problem of bootstrapping trust in commodity computers is thoroughly discussed by Parno et al. [65]. The authors present a survey of related work covering existing approaches and techniques to validate the integrity of local and remote computers. This includes "mechanisms for securely collecting and storing information about the execution environment", a goal for which secure coprocessors are used to achieve. Here, the compu-

[9] See also [99].

tation of cryptographic hashes over software binaries, including all referenced libraries at a point of time where the software has not yet booted up, is one example. The technique of a trusted boot was introduced by Gasser et al. [52] in 1989. Here, "a root of trust initiates the chain of trust", which means that the system's BIOS (basic input/output system) checks and initializes the bootloader while the bootloader again checks and initializes the operating system.

The survey from Parno et al. clearly shows the importance of secure coprocessors for commodity computer systems, as the trust with respect to the CPU is very limited, requiring the application of additional hardware. Next to secure coprocessors such as the IBM 4758, trusted platform modules (TPM) [90] are commonly used to validate an information system's[10] integrity. In contrast to external auxiliary processors, onboard modules are intrinsically not as immune to physical attacks. Parno et al. also discuss the limitations of the presented real-world and in-research approaches of bootstrapping trust, which mainly target the degree to which the corresponding technique can guarantee untampered hard- and software. The authors name two weaknesses, i.e., the shortcomings of load-time-only guarantees and the threat of physical hardware attacks. The former stresses the fact that load-time techniques can validate that the loaded software has not been modified, but cannot prevent attackers from, for instance, exploiting a buffer overflow vulnerability inside said software. The latter mainly focuses on the physical vulnerability of TPM chips, which ranges from grounding the module's reset pin to simply removing the entire chip. The survey by Parno et al. reveals that coprocessor-based techniques are among the most promising approaches in order for establishing trust in commodity computers, mainly because of the architecture and security features of such special-purpose computing entities.

Bibliography

[1] AHO, A. V., AND CORASICK, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM 18*, 6 (June 1975), 333–340. doi:10.1145/360825.360855. 28

[2] ALHERBISH, J. The BBN Butterfly Family: An Overview. In *High-Performance Computing and Networking*, W. Gentzsch and U. Harms, Eds., vol. 797 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994, pp. 489–490. doi:10.1007/3-540-57981-8_166. 19

[3] ANDRES, S., AND KENYON, B. *Security Sage's Guide to Hardening the Network Infrastructure*. Syngress, May 2004. 26

[4] BACKUS, J. Can Programming be Liberated from the von Neumann style?: A Functional Style and its Algebra of Programs. *Communications of the ACM 21*, 8 (Aug. 1978), 613–641. doi:10.1145/359576.359579. 18

[5] BAKER, Z., AND PRASANNA, V. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)* (Napa, CA, USA, Apr. 2004), IEEE Computer Society, pp. 135–144. doi:10.1109/FCCM.2004.6. 27

[6] BORKAR, S. Y., DUBEY, P., KAHN, K. C., KUCK, D. J., MULDER, H., PAWLOWSKI, S. S., AND RATTNER, J. R. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. White Paper, Intel Corporation, 2005. (Last checked: August 22, 2012). 18

[10] This includes computers of all sizes, ranging from smartphones to mainframes.

[7] CARRIER, B. D., AND GRAND, J. A Hardware-based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation 1*, 1 (Feb. 2004), 50–60. doi:10.1016/j.diin.2003.12.001. 22

[8] CHO, Y., AND MANGIONE-SMITH, W. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)* (Napa, CA, USA, Apr. 2004), IEEE Computer Society, pp. 125–134. doi:10.1109/FCCM.2004.25. 27

[9] CHO, Y. H., AND MANGIONE-SMITH, W. H. A Pattern Matching Coprocessor for Network Security. In *Proceedings of the 42nd Annual Design Automation Conference (DAC 2005)* (Anaheim, CA, USA, June 2005), ACM, pp. 234–239. (Last checked: August 22, 2012). 27, 67

[10] CHO, Y. H., NAVAB, S., AND MANGIONE-SMITH, W. H. Specialized Hardware for Deep Network Packet Filtering. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002)* (Montpellier, France, Sept. 2002), M. Glesner, P. Zipf, and M. Renovell, Eds., vol. 2438 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 452–461. doi:10.1007/3-540-46117-5_48. 26

[11] CLARK, C. R., AND SCHIMMEL, D. E. Scalable Pattern Matching for High Speed Networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)* (Napa, CA, USA, Apr. 2004), IEEE Computer Society, pp. 249–257. doi:10.1109/FCCM.2004.50. 27

[12] CLARK, P. C., AND HOFFMAN, L. J. BITS: A Smartcard Protected Operating System. *Communications of the ACM 37*, 11 (Nov. 1994), 66–70. doi:10.1145/188280.188371. 21

[13] COOK, D. L., IOANNIDIS, J., KEROMYTIS, A. D., AND LUCK, J. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *Proceedings of the the Cryptographers' Track at the RSA Conference (CT-RSA 2005)* (San Francisco, CA, USA, Feb. 2005), A. Menezes, Ed., vol. 3376 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 334–350. doi:10.1007/978-3-540-30574-3_23. 22, 93

[14] CROSBIE, M., DOLE, B., ELLIS, T., KRSUL, I., AND SPAFFORD, E. IDIOT – Users Guide. Tech. Rep. TR-96-050, Purdue University, Sept. 1996. (Last checked: August 22, 2012). 30

[15] DHAMDHERE, D. M. *Operating Systems: A Concept-based Approach*, 2nd ed. McGraw Hill, May 2006. 18

[16] DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T., AND LOCKWOOD, J. Deep Packet Inspection using Parallel Bloom Filters. In *Proceedings of the 11th Symposium on High Performance Interconnects (HIS 2003)* (Saint Louis, MO, USA, Aug. 2003), IEEE Computer Society, pp. 44–51. doi:10.1109/CONECT.2003.1231477. 26

[17] DOIG, L., BREWER, N., AND 'KWIGGINS'. Howard Aiken, Sept. 2008. (Last checked: August 22, 2012). 18

[18] DOLMATOV, V., ED. GOST 28147-89: Encryption, Decryption, and Message Authentication Code (MAC) Algorithms. RFC, Cryptocom, Mar. 2010. (Last checked: August 22, 2012). 22

[19] DORNSEIF, M. 0wned by an iPod, Nov. 2004. (Last checked: August 22, 2012). 22, 69

[20] DUFLOT, L., PEREZ, Y.-A., VALADON, G., AND LEVILLAIN, O. Can you still trust your network card? In *11th Canada Security West (CanSecWest 2010)* (Vancouver, Canada, Mar. 2010). (Last checked: August 22, 2012). 30

[21] FECHNER, B. GPU-Based Parallel Signature Scanning and Hash Generation. In *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS 2010)* (Hannover, Germany, Feb. 2010), C. Müller-Schloer, W. Karl, and S. Yehia, Eds., vol. 5974 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 133–138. (Last checked: August 22, 2012). 24

[22] FLYNN, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers C-21*, 9 (Sept. 1972), 948–960. doi:10.1109/TC.1972.5009071. 18, 20

[23] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A Sense of Self for Unix Processes. In *Proceedings of the 17th IEEE Symposium on Security and Privacy (S&P 1996)* (Oakland, CA, USA, May 1996), IEEE Computer Society, pp. 120–128. doi:10.1109/SECPRI.1996.502675. 24

[24] GEE, A. Research into GPU Accelerated Pattern Matching for Applications in Computer Security. Tech. rep., University of Canterbury, Nov. 2009. (Last checked: August 22, 2012). 29

[25] GILREATH, W. F., AND LAPLANTE, P. A. *Computer Architecture: A Minimalist Perspective*, 1st ed. Springer-Verlag, Mar. 2003. 17

[26] GOLDSTINE, A. A Report on the ENIAC. Tech. Rep. 1, University of Pennsylvania, June 1946. (Last checked: August 22, 2012). 18

[27] GOLDSTINE, H. H. *The Computer: from Pascal to von Neumann*. Princeton University, Oct. 1972. 18

[28] GÜBELI, R., KÄSER, H., KLAUS, R., AND MÜLLER, T. *Technische Informatik II – Mikroprozessor-Hardware und Programmiertechniken*, 2nd ed. vdf Hochschulverlag, Feb. 2010. 18

[29] HANDSCHUH, H., AND PAILLIER, P. Smart Card Crypto-Coprocessors for Public-Key Cryptography. In *Smart Card Research and Applications*, J.-J. Quisquater and B. Schneier, Eds., vol. 1820 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000, pp. 372–379. doi:10.1007/10721064_35. 21

[30] HARRISON, O., AND WALDRON, J. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)* (Vienna, Austria, Sept. 2007), P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 209–226. doi:10.1007/978-3-540-74735-2_15. 22, 93

[31] HELLIGE, H. D. *Geschichten der Informatik: Visionen, Paradigmen, Leitmotive*, 1st ed. Springer-Verlag, 2004. 17

[32] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006. 8, 18, 44, 46, 47, 48, 49

[33] HOOT, S. GPGPU: The Evolution of the Coprocessor. Tech. rep., Sage Electronic Engineering, May 2009. (Last checked: August 22, 2012). 20

[34] HORSPOOL, R. N. Practical Fast Searching in Strings. *Software: Practice and Experience 10*, 6 (June 1980), 501–506. doi:10.1002/spe.4380100608. 28

[35] HUANG, N.-F., CHU, Y.-M., AND HSU, H.-W. Graphics Processor-based High Performance Pattern Matching Mechanism for Network Intrusion Detection. In *Intrusion Detection Systems*, P. Skrobanek, Ed. InTech, Mar. 2011, ch. 16, pp. 287–306. (Last checked: August 22, 2012). 29

[36] HUANG, N.-F., HUNG, H.-W., LAI, S.-H., CHU, Y.-M., AND TSAI, W.-Y. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINAW 2008)* (Okinawa, Japan, Mar. 2008), IEEE Computer Society, pp. 62–67. doi:10.1109/WAINA.2008.145. 28, 29, 93

[37] HUGHES, C., AND HUGHES, T. *Professional Multicore Programming: Design and Implementation for C++ Developers*. John Wiley & Sons, Aug. 2008. 18

[38] HUTCHINGS, B. L., FRANKLIN, R., AND CARVER, D. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)* (Napa, CA, USA, Apr. 2002), J. Arnold and K. L. Pocek, Eds., IEEE Computer Society, pp. 111–120. doi:10.1109/FPGA.2002.1106666. 26

[39] IBM. IBM PCIe Cryptographic Coprocessor. (Last checked: August 22, 2012). 22

[40] IBM. IBM's ASCC Introduction. (Last checked: August 22, 2012). 18

[41] JACOB, N., AND BRODLEY, C. Offloading IDS Computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)* (Miami Beach, FL, USA, Dec. 2006), IEEE Computer Society, pp. 371 –380. doi:10.1109/ACSAC.2006.35. 28, 92

[42] KIM, B., YOON, S., AND OH, J. ATPS – Adaptive Threat Prevention System for High-Performance Intrusion Detection and Response. In *Proceedings of the 10th Asia-Pacific Network Operations and Management Symposium (APNOMS 2007)* (Sapporo, Japan, Oct. 2007), S. Ata and C. S. Hong, Eds., vol. 4773 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 344–353. doi:10.1007/978-3-540-75476-3_35. 27

[43] KIM, B., YOON, S., AND OH, J. Multi-hash based Pattern Matching Mechanism for High-Performance Intrusion Detection. *International Journal of Computers 3*, 1 (2009), 115–124. (Last checked: August 22, 2012). 27

[44] KNUTH, D. E., MORRIS, J. J. H., AND PRATT, V. R. Fast Pattern Matching in Strings. *SIAM Journal on Computing 6*, 2 (1977), 323–350. doi:10.1137/0206024. 25, 28

[45] KOROBITSIN, V., AND ILYIN, S. GOST-28147 Encryption Implementation on Graphics Processing Units. In *Proceeding of the 3rd International Conference on Availability, Reliability and Security (ARES 2008)* (Barcelona , Spain, Mar. 2008), IEEE Computer Society, pp. 967–974. doi:10.1109/ARES.2008.103. 22

[46] KOUZINOPOULOS, C., AND MARGARITIS, K. String Matching on a Multicore GPU Using CUDA. In *Proceedings of the 13th Panhellenic Conference on Informatics (PCI 2009)* (Corfu, Greece, Sept. 2009), IEEE Computer Society, pp. 14–18. doi:10.1109/PCI.2009.47. 28

[47] KUPERMAN, B. A. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. Ph.D. thesis, Purdue University, Aug. 2004. (Last checked: August 22, 2012). 30

[48] LEHTI, R., AND VIROLAINEN, P. AIDE (Advanced Intrusion Detection Environment), 1999. (Last checked: August 22, 2012). 23

[49] LI, Y., MCCUNE, J. M., AND PERRIG, A. VIPER: Verifying the Integrity of PERipherals' Firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)* (Chicago, IL, USA, Oct. 2011), ACM, pp. 3–16. doi:10.1145/2046707.2046711. 30

[50] LIBSTER, E., AND KORNBLUM, J. D. A Proposal for an Integrated Memory Acquisition Mechanism. *ACM SIGOPS Operating Systems Review 42*, 3 (Apr. 2008), 14–20. doi:10.1145/1368506.1368510. 23

[51] LOPRESTI, D. P. Rapid Implementation of a Genetic Sequence Comparator using Field-Programmable Logic Arrays. In *Proceedings of the 13th University of California/Santa Cruz Conference on Advanced research in VLSI (ARVLSI 1991)* (Santa Cruz, CA, USA, Mar. 1991), C. H. Séquin, Ed., MIT Press, pp. 138–152. (Last checked: August 22, 2012). 25

[52] M. GASSER, A. GOLDSTEIN, C. K., AND LAMPSON, B. The Digital Distributed System Security Architecture. In *Proceedings of the 12th National Computer Security Conference (NCSC 1989)* (Baltimore, MD, USA, Oct. 1989). 31

[53] MAJO, Z., AND GROSS, T. R. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR 2011)* (Haifa, Israel, May 2011), ACM, pp. 12:1–12:10. doi:10.1145/1987816.1987832. 19

[54] MARZIALE, L., III, G. G. R., AND ROUSSEV, V. Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. *Digital Investigation 4* (Sept. 2007), 73–81. doi:10.1016/j.diin.2007.06.014. 22, 93

[55] MCCALPIN, J. D. Sustainable Memory Bandwidth in Current High Performance Computers, Oct. 1995. (Last checked: August 22, 2012). 24

[56] MCEVOY, T. R., AND WOLTHUSEN, S. D. Host-Based Security Sensor Integrity in Multiprocessor Environments. In *Proceedings of the 6th Information Security Practice and Experience Conference (ISPEC 2010)* (Seoul, Korea, May 2010), J. Kwak, R. Deng, Y. Won, and G. Wang, Eds., vol. 6047 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 138–152. doi:10.1007/978-3-642-12827-1_11. 24, 43, 44, 53, 54, 57, 58, 59

[57] MOLINA, J. Using Embedded Auditors for Intrusion Detection Systems. Master's thesis, University of Maryland, 2001. (Last checked: August 22, 2012). 23, 24

[58] MOLINA, J., AND ARBAUGH, W. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security (ICICS 2002)* (Singapore, Dec. 2002), R. Deng, F. Bao, J. Zhou, and S. Qing, Eds., vol. 2513 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 291–302. doi:10.1007/3-540-36159-6_25. 6, 8, 23, 24, 25, 58

[59] MOSCOLA, J., LOCKWOOD, J., LOUI, R., AND PACHOS, M. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)* (Napa, CA, USA, Apr. 2003), IEEE Computer Society, pp. 31–38. doi:10.1109/FPGA.2003.1227239. 26

[60] MÜLLER-SCHLOER, C. *RISC-Workstation-Architekturen: Prozessoren, Systeme und Produkte*. Springer-Verlag, 1991. 20

[61] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)* (Seattle, WA, USA, Mar. 2008), S. Eggers and J. Larus, Eds., ACM, pp. 308–318. doi:10.1145/1353535.1346321. 30, 43

[62] NTT COMMUNICATIONS. NTT Com's Japan-U.S. Backbone Bandwidth Reaches 400 Gbps, Jan. 2011. (Last checked: August 22, 2012). 25

[63] NVIDIA CORPORATION. CUDA 1.0, June 2007. (Last checked: August 22, 2012). 28

[64] NVIDIA CORPORATION. NVIDIA CUDA Programming Guide 2.2.1, May 2009. (Last checked: August 22, 2012). 20, 50

[65] PARNO, B., MCCUNE, J. M., AND PERRIG, A. Bootstrapping Trust in Commodity Computers. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)* (Berkeley, CA, USA, May 2010), IEEE Computer Society, pp. 414–429. doi:10.1109/SP.2010.32. 30, 94

[66] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[67] PRYOR, D. V., THISTLE, M. R., AND SHIRAZI, N. Text searching on Splash 2. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, USA, Apr. 1993), IEEE Computer Society, pp. 172–177. doi:10.1109/FPGA.1993.279466. 25

[68] RAMANATHAN, J., AND NI, L. Critical Factors in NUMA Memory Management. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS 1991)* (Arlington, TX, USA, May 1991), IEEE Computer Society, pp. 500–507. doi:10.1109/ICDCS.1991.148717. 19

[69] RANKL, W., AND EFFING, W. *Smart Card Handbook*, 3rd ed. Wiley, 2004. 21

[70] ROESCH, M. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on Large Installation Systems Administration (LISA 1999)* (Seattle, WA, USA, Nov. 1999), USENIX Association, pp. 229–238. (Last checked: August 22, 2012). 26

[71] ROSENBERG, U. Using Graphic Processing Unit in Block Cipher Calculations. Master's thesis, Iniversity of Tartu, 2007. (Last checked: August 22, 2012). 22

[72] SEEGER, M. M. Internetkonnektivität als Indikator für wirtschaftliche Stärke – Das Internet in Vergangenheit und Gegenwart. *Wirtschaftsinformatik & Management*, 2/2011 (Feb. 2011), 46–50. 25

[73] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[74] SEEGER, M. M., AND WOLTHUSEN, S. D. Towards Concurrent Data Sampling using GPU Coprocessing. In *Proceedings of the 6th International Workshop on Secure Software Engineering (SecSE 2012)* (Prague, Czech Republic, Aug. 2012), IEEE Computer Society. 6, 10, 12, 28

[75] SIDHU, R., AND PRASANNA, V. K. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)* (Rohnert Park, CA, USA, Apr. 2001), IEEE Computer Society, pp. 227–238. doi:10.1109/FCCM.2001.22. 25, 26

[76] SIDHU, R. P. S., MEI, A., AND PRASANNA, V. K. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 1999)* (Monterey, CA, USA, Feb. 1999), ACM, pp. 217–226. doi:10.1145/296399.296463. 25

[77] SKROBANEK, P., Ed. *Intrusion Detection Systems*. InTech, Mar. 2011. (Last checked: August 22, 2012). 29

[78] SMITH, S. W. Secure Coprocessing Applications and Research Issues. Tech. Rep. LA-UR-96-2805, Los Alamos National Laboratory, Aug. 1996. (Last checked: August 22, 2012). 19, 94

[79] SMITH, S. W., PALMER, E. R., AND WEINGART, S. Using a High-Performance, Programmable Secure Coprocessor. In *Proceedings of the 2nd International Conference on Financial Cryptography (FC 1998)* (Anguilla, British West Indies, Feb. 1998), R. Hirschfeld, Ed., vol. 1465 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 73–89. doi:10.1007/BFb0055468. 19

[80] SMITH, S. W., AND WEINGART, S. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks 31*, 8 (Apr. 1999), 831–860. doi:10.1016/S1389-1286(98)00019-X. 19, 23, 25

[81] SOURDIS, I. Efficient and High-Speed FPGA-based String Matching for Packet Inspection. Master's thesis, Technical University of Crete, July 2004. (Last checked: August 22, 2012). 27

[82] SOURDIS, I., AND PNEVMATIKATOS, D. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Proceedings of the 13th International Conference on Field Programmable Logic and Application (FPL 2003)* (Lisbon, Portugal, Sept. 2003), P. Y. K. Cheung and G. A. Constantinides, Eds., vol. 2778 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 880–889. doi:10.1007/978-3-540-45234-8_85. 26, 27

[83] SOURDIS, I., PNEVMATIKATOS, D., AND VASSILIADIS, S. An Evaluation of FPGA-based IDS Pattern Matching Techniques. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRisc 2005)* (Veldhoven, Netherlands, Nov. 2005), pp. 449–453. (Last checked: August 22, 2012). 27

[84] SOURDIS, I., PNEVMATIKATOS, D., AND VASSILIADIS, S. Scalable Multigigabit Pattern Matching for Packet Inspection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 2 (Feb. 2008), 156–166. doi:10.1109/TVLSI.2007.912036. 27

[85] SPEISER, A. P. *Digitale Rechenanlagen. Grundlagen, Schaltungstechnik, Arbeitsweise, Betriebssicherheit*. Springer-Verlag, 1961. 17

[86] SUNDAY, D. M. A Very Fast Substring Search Algorithm. *Communications of the ACM 33*, 8 (Aug. 1990), 132–142. doi:10.1145/79173.79184. 28

[87] THE SHMOO GROUP. DefCon Capture the Flag Contest traces, Sept. 2001. (Last checked: August 22, 2012). 28

[88] THOMPSON, R. B., AND THOMPSON, B. F. *PC Hardware in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, 2003. 21

[89] TRENT, G., AND SAKE, M. WebSTONE: The First Generation in HTTP Server Benchmarking, Feb. 1995. (Last checked: August 22, 2012). 24

[90] TRUSTED COMPUTING GROUP. TPM Main Specification. Technical Specification Version 1.2, Revision 116, Trusted Computing Group, Mar. 2011. (Last checked: August 22, 2012). 31

[91] TUMEO, A., SECCHI, S., AND VILLA, O. Experiences with String Matching on the Fermi Architecture. In *Proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS 2011)* (Como, Italy, Feb. 2011), M. Berekovic, W. Fornaciari, U. Brinkschulte, and C. Silvano, Eds., vol. 6566 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 26–37. doi:10.1007/978-3-642-19137-4_3. 29

[92] TURING, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *The London Mathematical Society s2–42*, 1 (1937), 230–265. doi:10.1112/plms/s2-42.1.230. 17

[93] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium of Recent Advances in Intrusion Detection (RAID 2008)* (Boston, MA, USA, Sept. 2008), vol. 5230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 116–134. doi:10.1007/978-3-540-87403-4_7. 28, 67, 92

[94] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. GPU-Assisted Malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)* (Nancy, France, Oct. 2010), IEEE Computer Society, pp. 1–6. doi:10.1109/MALWARE.2010.5665801. 29, 93

[95] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. Parallelization and Characterization of Pattern Matching using GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC 2011)* (Austin, TX, USA, Nov. 2011), IEEE Computer Society. doi:10.1109/IISWC.2011.6114181. 29

[96] VON NEUMANN, J. First Draft of a Report on the EDVAC. Tech. Rep. W-670-ORD-492, University of Pennsylvania, June 1945. doi:10.1109/85.238389. 17, 20, 23

[97] WILLIAMS, P., AND SPAFFORD, E. CuPIDS Enhances StUPIDS: Exploring a Co-Processing Paradigm Shift in Information System Security. In *Proceedings of the 6th Annual IEEE SMC Information Assurance Workshop (IAW 2005)* (West Point, NY, USA, June 2005), IEEE Computer Society, pp. 126–133. doi:10.1109/IAW.2005.1495943. 29

[98] WILLIAMS, P. D. *CuPIDS: Increasing Information System Security through the use of Dedicated Co-Processing*. Ph.D. thesis, Purdue University, Aug. 2005. (Last checked: August 22, 2012). 29

[99] WILLIAMS, P. D., AND SPAFFORD, E. H. CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection. *Computer Networks 51*, 5 (Apr. 2007), 1284–1298. doi:10.1016/j.comnet.2006.09.011. 30, 43, 67, 93, 94

[100] WU, C., YIN, J., CAI, Z., ZHU, E., AND CHEN, J. A Hybrid Parallel Signature Matching Model for Network Security Applications Using SIMD GPU. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies (APPT 2009)* (Rapperswil, Switzerland, Aug. 2009), Y. Dou, R. Gruber, and J. Joller, Eds., vol. 5737 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–204. doi:10.1007/978-3-642-03644-6_15. 28

[101] WU, S., AND MANBER, U. A Fast algorithm for Multi-Pattern Searching. Tech. rep., The University of Arizona, 1994. (Last checked: August 22, 2012). 28, 93

[102] WULF, W. A., AND MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News 23*, 1 (Dec. 1994), 20–24. doi:10.1145/216585.216588. 18

[103] YEE, B. S. *Using Secure Coprocessors*. Ph.D. thesis, Carnegie Mellon University, May 1994. (Last checked: August 22, 2012). 19

[104] YEE, B. S., AND TYGAR, J. D. Secure Coprocessors in Electronic Commerce Applications. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce (EC 1995)* (New York, NY, USA, July 1995), USENIX Association, pp. 155–170. (Last checked: August 22, 2012). 19

[105] ZHANG, X., AND QIN, X. Performance Prediction and Evaluation of Parallel Processing on a NUMA Multiprocessor. *IEEE Transactions on Software Engineering 17*, 10 (Oct. 1991), 1059–1068. doi:10.1109/32.99193. 19

[106] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure Coprocessor-Based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, July 2002), ACM, pp. 239–242. doi:10.1145/1133373.1133423. 23, 24, 43, 53, 54, 59, 61, 62, 67

Part II

Publications

Chapter 3

Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency

Abstract

Whilst the precise objectives and mechanisms used by malicious code will vary widely and may involve wholly unknown techniques to achieve their respective objectives, certain second-order operations such as privilege escalation or concealment of the code's presence or activity are predictable. In particular, concealment mechanisms must modify well-known data structures, which could be detected trivially otherwise. We argue that any such mechanism is necessarily non-atomic and can hence be detected through concurrent observations forcing an interleaved linearization of the malicious code with observations of memory state changes induced in tightly coupled concurrent processing units. Extending previous research for the case of symmetric concurrent observation, we propose a computational model and observation mechanism for the case of tightly coupled asymmetric concurrent processing units as may be found in most current computing environments with particular emphasis on metrics for the cost of forced synchronizations and resource contention caused by observations. We argue that the resulting observations will provide a novel sensor datum for intrusion detection but may also be used as a standalone probabilistic detection mechanism particularly suited to detect attacks in progress.

3.1 Introduction

To achieve its objectives, malicious code will typically have to perform some form of privilege escalation and must also evade detection, potentially also in a persistent manner. The techniques used by such code are not all known a priori, and cannot be prevented entirely as new approaches may not be caught by analytical approaches [10, 4]. Moreover, malicious code will be able to reconstruct the legitimate state of the target system or construct a plausible alternative state in which its presence is effectively concealed. Parallel monitoring of execution has been a long-established approach in fault-tolerant systems and has also been proposed for use in commodity systems based on multithreading mechanisms by Oplinger and Lam [13], although without consideration of security concerns; similarly, the use of low-level speculative execution and parallelization of security checks on multiple concurrent processing units proposed by Nightingale *et al.* does not provide protection of the monitoring system itself [11]. Coprocessor-based detection is the topic of [19] and [14]. In contrast to [19] who considers an intrusion detection system as self-protected as soon as it does not run on the host [9] introduced a mechanism which is distinguished by distributed computations hard to subvert even with full knowledge of this mechanism. Loosely coupled multiprocessing or coprocessor architectures for execution monitoring and execution monitoring itself have also been described by Williams and Spafford [18]; their mechanism uses the facilities and capabilities afforded by a general purpose symmetrical multi-processing (SMP) computer architecture, dedicating one or more CPUs exclusively to security-related tasks, resulting in a relatively high overhead. Whilst the *number*

of potential attacks and vulnerabilities is difficult to bound, some of their actions are constrained (cf. [9]), and require subverting certain functions. Petroni et al. (cf. [15]) introduce an architecture to detect malicious modifications of kernel memory by comparing the actual observed kernel state with a specification of a kernel state known to be correct, describing a mechanism to detect semantic integrity violations in dynamic kernel data structures while in [16] they introduce a monitoring system concerned with observing the operating system kernel integrity, based on a property called *state-base control-flow integrity*, demonstrating that it is indeed possible to spot an intrusion by observing only certain parts of kernel and memory structure. Finally, any such sensors can be fused as, e.g., proposed by Almgren et al. (cf. [3]) using a multi-sensor model to improve automated attack.

The remainder of this paper is structured as follows: Section 3.2 introduces our computational model which is founded on the hierarchical arrangement found in modern computers. Also in this section, a cost model considering the case of asymmetrical multiprocessing environments tightly coupled through shared memory systems and an example of working set interference is presented. Section 3.3 then proposes an interference model – where *interference* is divided into demand and coherence related portions – and two equations for calculating the overall *average memory access time* and the *interference ratio*. In Section 3.4, we present our conclusion and future work.

3.1.1 Notation

Table 3.1 provides an overview of the notation used throughout this paper. We divide our notations into three groups: The first group (Components) is the subject of our model. With the second and third group, we pool notations with regard to Memory Access and Cache Performance, respectively.

3.2 Computational Model

Industry-standard computer systems consist of several different components, serving several different purposes. Some of the components located on a main board are directly interconnected and even share certain memory locations. When performing observations (read operations) or write operations, which can cause synchronization protocols (cf. [7]) to be applied, it must be taken into account that these interconnected components operate at substantially different speeds. In approximate order of speed the component model relevant for us includes one or more CPUs with each CPU core maintaining a separate cache structure (typically one, possibly two-level). A cache may be shared between processing cores, or be connected independently [7]. In the former case, synchronization will typically be slower than for the first level of caching. The next substantial level of speed difference manifests itself at the interface between CPU and main memory. For the purpose of this discussion it is most relevant to note the significant differences in speed between on-CPU memory (which, as noted above, may also itself have several levels of differing speeds) and main memory.

3.2.1 Cost Model

We argue that malicious code intended to be executed by one of the central processing units will, at some point in its life-cycle, come to reside either entirely or partially in a shared main memory of a computer system, where it may be visible and accessible by other entities connected to said memory. As shown in [8], it is possible to either trivially detect known or inferred signatures of malicious code or to probabilistically detect activity patterns associated with malicious behavior *using concurrent observation mechanisms*. As most malicious code will attempt to obfuscate its presence and must respond to a set of well-known detection mechanisms, several activity patterns to be searched for are hence sufficiently

Table 3.1: Overview of notations used throughout this paper.

Components

P	Processing unit (P_i (normal duty), P_j (observation))
M	Memory component (e.g., first level cache or physical memory)
G	Graph consisting of P- and M-vertices
S	Process

Memory Access

m^l	a set of one or more memory locations within the l^{th} M-vertex
$[m^l]$	data stored in m^l
$[m^l]'$	altered data stored in m^l
m_i^l	memory access of memory location i in M-vertex l
c	costs (measured in clock cycles)
$c(m_i^l)$	costs of m_i^l
$c(m^l)$	costs accruing due accessing memory locations m in the l^{th} M-vertex
k	computation
P_i^k	computation k executed by processing unit P_i
$m^l_{P_i^k}$	memory locations m belonging to the l^{th} M-vertex and are altered by P_i^k

Cache Performance

$W(t,T)$	Working set of pages T (T is a set) at point in time t
h_i^t	hit time to the i^{th} M-vertex
ms_i^r	miss rate to the i^{th} M-vertex
ms_i^{pd}	miss penalty to the i^{th} M-vertex (demand-related)
ms_i^{pc}	miss penalty to the i^{th} M-vertex (coherent-related)
X_t	average memory access time
t_{opt}	time needed/used for exectuting a process under optimal conditions
t_{norm}	time needed/used for exectuting a process under normal conditions
$\|A\|, \|B\|$	sets of memory locations
$\overline{ac'b'def}$	a specific cache line
$\overline{c'b'de}$	a specific working set
$P_i \overset{page}{\rightarrow}$	Processing unit i writing to a set of pages
$P_i \overset{page}{\leftarrow}$	Processing unit i reading from a set of pages

constrained to allow identification of manipulations of *semantic choke points*[1] based on a restricted set of data structures and control-flows and in-memory side effects of such control flows. The observation model described in [8] considered the case of symmetric multiprocessing environments, which also exhibit the key properties of non-atomic operations of observed (malicious) code execution resulting in a set of possible *linearizations* of memory accesses forced by the underlying computing substrate. However, in the following we are extending these results by considering the case of asymmetrical multiprocessing environments tightly coupled through shared memory systems, which have become ubiquitous designs at all scales of computing environments from smartphones to mainframe systems as both performance and efficiency considerations make this approach compelling. Such environments are characterized by having multiple processing units P coupled by shared memory entities M. We model the interconnection between these entities by constructing a vertex-colored (planar) graph G in which a P-vertex is never connected to another P-vertex by a direct edge (i.e., allowing for cascaded cache structures). We note that in most computation environments, this results in a finite tree (i.e., G is connected and acyclical) in which the main memory is the root vertex; however, as configurations with cycles exist, we are considering the general case of a planar graph as described above. We define that an *observation* of the root vertex is possible where two P-vertices P_i and P_j are connected by an M-path, i.e., a path in the graph consisting only of M-vertices; its constraint on the

[1] "Data structures and, more generally relations and invariants over data structures which subversion mechanisms must manipulate to conceal their presence as a failure to do so would result in their being detectable by existing mechanisms."[8]

model is motivated by the fact that direct determination of the internal state of a processing unit will rarely be possible. A computation k on processing unit P_i is designated as P_i^k and may alter the states of one or more memory locations m^l on the l^{th} M-vertices directly connected to P_i; the set of all memory locations which may be altered by P_i^k is hence abbreviated as $m^l_{P_i^k}$ where l is the index of the M-vertex. To characterize the costs incurred by an observation, we now associate a cost c with each memory access m_i^l where i is the index of the accessed memory location within the set of all memory locations m^l and l the M-vertex index, respectively. However, a key aspect of our model is to observe that the cost $c(m_i^l)$ is not necessarily constant owing to the relation forced on multiple memory locations (i.e., caching) described above where the memory locations of the target computation k to be observed are within the working set (cf. Section 3.2.2) and a M-connected subgraph exists between observer computation (P_j-vertex) and observation target computation (P_i-vertex). We note that $c(m^l) \ll c(m^{l+1})$ in such a path, i.e., the cost (speed) of accessing each (even adjacent) memory hierarchy will potentially differ by one or more orders of magnitude.

As noted before, the observation costs are not necessarily constant. This is especially true, when the observed memory locations are part of a working set and are altered by P_i. [5] defines a working set as the set of most recently referenced pages, where the term *pages* could just be replaced by any other unit of information: $W(t,T)$, therefore, is a working set of distinct and most recently referenced pages at time t, i.e., intuitively, the smallest subset of pages that must reside in main memory such that a program operates at some desired level of efficiency [6]. When a computation k is executed by P_i the affected memory locations m^l are part of a working set, which affects the execution performance positively in general. Pages can get replaced of course, for example due to capacity conflicts. Whether or not a page being part of a working set will be replaced by another is most commonly decided by using the least recently used (LRU) scheme [7], where the data that has not been used for the longest time gets replaced. We assume that whenever a page gets replaced this happens for the benefit of efficiency. When observing M_n using P_j, there exists a non-negligible probability that these memory locations are referenced throughout the M-connected graph all the way up to M_i, potentially affecting performance if those referenced locations are altered whilst being subject of an observation. For altered memory locations a write-back strategy takes care of the coherence, as due to performance reasons the changes made are not instantly carried out to the bottom (or any lower level M-vertex) of the hierarchy. This is only done when certain conditions hold (cf. [7]). Important for now is the condition that the original data ($[m^l], l = n$ residing in M_n)[2] of a referenced and altered memory location ($[m^l]', l = i$ residing in M_i) is observed by P_j. In this case a forced synchronizations take place. Usually write-back strategies pay attention to what is to be written back to M_n and when. But a write-back can also be triggered. Particularly by accessing $[m^l], l = n$ when there exists $[m^l]', l = i$ which causes $[m^l]'$ to be written back to the memory location of $[m^l]$. Whenever our observation causes a forced write-back, we generate traffic and, therefore, costs. This is especially critical if $[m^l]'$ is part of a working set, as we will explain now.

3.2.2 Example for Working Set Interference

Our initial working set $W(t,T) = \overline{bcde}$ is part of the following cache line: $a\overline{bcde}f$
The value at the very left of the working set (b) denotes, that b was accessed most recently of all pages, while the access of the value at the very right (e) has been furthermost in the past and a and f are not part of the working set. When a processor accesses a page in order to write to it, we denote this as $P_i \overset{page}{\rightarrow}$ and when it reads, we write $P_i \overset{page}{\leftarrow}$ respectively, were the term $page$ is a set of pages containing at least one page.

[2] $[m^l]$ denotes data stored in m^l

Presuming the above described working set, after processing unit 1 wrote new data to pages b and c, expressed as $P_1 \overset{b,c}{\rightarrow}$ the cache line – the working set, to be precise – changed and looks like this: $\overline{ac'b'de}f$ with the apostrophe (′) denoting that those pages contain altered data that is not yet written back. After $P_1 \overset{d,e}{\leftarrow}$ (processing unit 1 reads pages d and e) we have the following structure: $\overline{aedc'b'}f$

Now our observation process starts, executed by processing unit 2 which is directly connected to the root vertex of our graph; namely the main memory. As P_1 and P_2 run independently and asynchronously, they can operate in parallel, possibly willing to consume the same data: $P_2 \overset{b}{\leftarrow} || P_1 \overset{b}{\leftarrow}$

Page b is still not synchronized with its original which causes forced synchronizations as soon as P_2 accesses the original and outdated copy residing in the main memory. Parallel to this, P_1 accesses its copy of b and has to wait, causing a wait situation and, therefore, costs (generally expressed as $c(m_i^l)$). Recall that the index l denotes the affiliation to the corresponding M-vertex and index i stands for the i^{th} memory location. Here, these costs apply not only once, but for every level of cache; the *further* a cache is away from the CPU the higher the costs. Again, presuming the altered cache line from above ($\overline{aedc'b'}f$) we now show what we exactly mean by the term of *interference*: Without any observation in place, $P_1 \overset{f}{\leftarrow}$ would evict page b from the working set, including now page f as the most recently accessed page: ($\overline{afedc'b'}$).

Imagining that we observed page b just before P_1 reads f, the final cache line would look like this:
$P_2 \overset{b}{\leftarrow}$: $\overline{abedc'}f$, $P_1 \overset{f}{\leftarrow}$: $a\overline{fbedc'}$
So, page c (still not written back) would have been evicted, instead of page b. This interferes the LRU schema, causing potentially higher costs, as pages get evicted that may still be useful. Also keep in mind that accessing the altered page b' in the last example causes a forced synchronization (but no wait this time).

3.3 Interference Model

When observing the M_n-vertex, the one at the bottom of our hierarchy (main memory), we are actually observing processes. We denote a process as S. The resources a process is built of (code) and works with (allocated memory) are loaded into M_n. As we do not know the exact memory access pattern of any/each process, we need a model to estimate the difference between the costs of the *normal duty* and those when our observation interferes with the *normal duty*. At some point in time, parts of any process make their way up the M_i-vertex and form working sets $W(t, T)$. It is important to notice that working sets are not necessarily disjoint. In other words a certain page can reside in one or more working sets, while there are possibly n working sets for n processes. The worst case scenario with regard to additional costs our observation can produce, occurs when a forced synchronization between top- and low-level M-vertices takes place; namely a cache and the main memory. For this to happen, two conditions must hold: 1) Data $[m^l]'$ residing in memory locations $m^l, l = i$ of M_i (cache) is an altered copy of the data $[m^l]$ in memory locations $[m^l], l = n$ of M_n (main memory), so $[m^l]' \neq [m^l]$. 2) m^l is accessed by either another M-vertex or a processing unit other than the one accessing $[m^l]'$. When $[m^l]$ is accessed by another M-vertex a protocol commonly known as *cache snooping* is concerned with providing each processing unit with the most current data. For example, $[m^{l,m}], l = i, m = j$ resides in two M-vertices $M_{i,j}$ on the same level, where i denotes the level and j the number; namely $M_{1,1}$ and $M_{1,2}$. Notice that $[m^{1,1}] = [m^{1,2}]$. Processing unit P_1 writes to $[m^{1,1}]$, which is stored in its directly connected M-vertex $M_{1,1}$, changing its data to $[m^{1,1}]'$. $M_{1,2}$ belonging to P_2 still contains $[m^{1,2}]$. As soon as P_2 tries to read $[m^{1,2}]$ a miss occurs, causing $[m^{1,2}]$ to get overwritten with $[m^{1,1}]'$. According to [7] the performance of a cache organization can

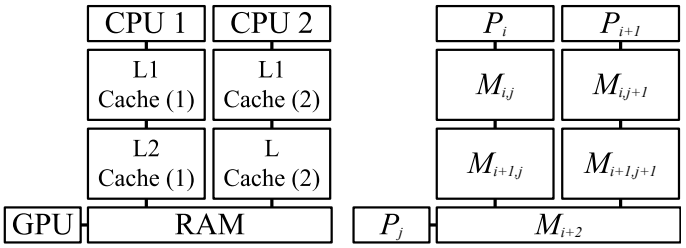


Figure 3.1: Graph G, with multiple P- and M-vertices on the same level. Left: A concrete component model. Right: The abstract component model.

be expressed in form of the *average memory access time* as depicted in Equation 3.1.

$$X_t = Hit\ time + Miss\ rate \cdot Miss\ penalty \tag{3.1}$$

$Hit\ time$ is the time to hit the cache and $Misspenalty$ is the time to replace the block from memory (i.e., the cost of a miss). To calculate the costs of a synchronization between multiple levels of M-vertices, as well as multiple M-vertices on the same level, we derive Equation 3.2 from what [7] suggests.

h_i^t = Hit time to the i^{th} M-vertex
ms_i^r = Miss rate of the i^{th} M-vertex
ms_i^{pd} = Miss penalty of the i^{th} M-vertex (demand-related)
ms_i^{pc} = Miss penalty of the i^{th} M-vertex (coherence-related)

$$X_t = \sum_{i=0}^{n} h_i^t + pd \cdot ms_i^r \cdot (h_{i+1}^t + ms_{i+1}^r \cdot ms_{i+1}^{pd}) + pc \cdot ms_i^r \cdot ms_i^{pc} \tag{3.2}$$

pd is a ratio between $0 \ldots 1$ expressing which part of the miss rate is demand-related, while pc is the ratio for coherence-related misses respectively. $pd + pc = 1$

The following hit times, miss rates and miss penalties are typical values for current desktop computers:

First level cache:
h_1^t = 1 cc (clock cycles)
ms_1^r = 5% mpi (misses per instruction)
ms_1^{pd} = 65 cc ($h_2^t + ms_2^r \cdot ms_2^{pd}$)
ms_1^{pc} = 3 cc

Second level cache:
h_2^t = 15 cc
ms_2^r = 25% mpi
ms_2^{pd} = 200 cc
ms_1^{pc} = 45 cc

Since coherence-related misses occur on the same level of cache, they have a rather small miss penalty. As a result of this, in this example they add only very little to the overall average memory access time. But with a lower level taken into account, the contribution of coherence-related misses rises. For illustrating the above stated equation we introduce a graphical representation of our component model with multiple P- and M-vertices on the same level in Figure 3.1.

We showed that with this generalization, we are able to not only calculate X_t for the occurrence of misses due to demanded data that is not available in M_i but in M_{i+1} but also to take into account that data may be incoherent in M-vertices on the same level. To consider this, the miss rate is divided into two fractions defined by pd and pc.

Regarding the additional costs generated by the observation we could now come up with a probabilistic model consisting of basically two probabilities: One, the probability that observed data is part of a working set and possibly altered by the corresponding processing unit. Two, the probability that we observe such data. We would end up with two

values hard to correlate. Furthermore, paying respect to coherence-related misses would make this model far too complex for our purpose:

Our observation can be seen as a totally passive process: We do not write to any M-vertex and we do not send instructions to any processing unit (besides P_j). The costs that accrue are due to the fact that during any synchronization of data the processing unit willing to compute has to wait until the synchronization is done before it can proceed. More formally, the higher costs appear, as P_i has to wait until the outdated data $[m^l, l = i]$ is synchronized with $[m^l]', l = n$.

So, what we need is a model that gives us a good measure of the computing time of a process under *normal duty* and under observation. We decided to build our measures on the paging and working set schemes, as those are two of the most powerful methods to overcome the ever rising gap between processor and main memory speed (cf. [2, 17]) and are fundamental when it comes to performance related issues.

The ideal conditions for any process to be executed in optimal time t_{opt} are a) all resources S fit into the working set $W(t, T)$ for any t and b) the resources are being consumed by one processing unit only. With these two conditions met demand-related (a), as well as coherence-related (b) misses would be eliminated. As this is very unlikely to happen – and if, only for very small processes – the normal conditions (t_{norm}, respectively) are described by a) the existence of a working set $W(t, T)$ which insures that the corresponding process can be executed at some level of efficiency and b) the resources are shared over two or more processing units in order to achieve a higher performance.

Presuming t_{norm} we are interested in the additional processing time occurring when the corresponding process is under observation. With regard to the equation mentioned earlier, it is the factor (or degree) by which the miss rate (m_i^r) of each memory hierarchy level is affected. One might say that the size of each memory level, as well as the set-associatively of each cache-level also pay a vital role when talking about interference; and that's true. But, as the named parameters are known for their impact, they are already reflected by the miss rate, as according to [7] a bigger cache size and higher set-associatively result in lower miss rates. For our purpose we define the following triple (r, w, x). Each variable can be expressed by a value between $0 \ldots 1$ (including 0 and 1): With regard to each working set, r is the percentage of read-only memory locations and w the percentage of the writeable ones where $r + w = 1$. All three values are time dependent, and must relate to the same period of time $[t]$: Choosing the length of the period of time is crucial as we will explain now: There is no such thing as a general memory access pattern for processes. While this may not be entirely true for scientific applications, we argue that it is true in general. But patterns can be derived for certain phases occurring during the life time of a process, e.g., the process start-up where due to an empty working set compulsory misses occur until the working set is built[3]. Whenever a new phase starts, the memory access pattern may change. Ideally the period of time regarding the triple (r, w, x) represents one phase. If phases are very short, it makes sense to let $[t]$ span several phases in order to relativize read or/and write peaks. Since the pure presence of a writeable memory location does not mean that it is actually being written, x stands for the percentage of processing time spent on really writing data (e.g., $x = 0.7$ means that 70% of time span $[t]$ for processing process S were used for writing data). The equation for calculating the interference ratio I is defined as follows:

$$I = \frac{w \cdot x}{(x - w) \cdot (r)} \cdot [t] \begin{cases} I \to \infty, & \text{for } w \to x \to 1, \\ I \to 0, & \text{for } x \to 0, \\ I = undef, & \text{for } t = 0 \end{cases} \tag{3.3}$$

[3] We know about the chance that the working set may never be entirely built, as it is possible that even during start-up time a process can massively renew its working set (mentioned in [1]), but leave this out of scope.

The leading fraction in Equation 3.3 is multiplied by the time span $[t]$ which serves the purpose of making clear that smaller time spans do not have as much influence on the result as bigger ones. The interference I approaches 0 for $x \to 0$: If no data is written, no inconsistency between the copied data in cache and the original data in physical memory exists. Therefore, no forced synchronizations will be triggered and so, no additional costs apply.

If $x \to 1$ data is written continuously, causing an ongoing forced synchronization if the process is under observation, which leads to a very high interference ratio, labeled as ∞. As mentioned before, $[t]$ is a time span and, therefore, cannot be 0.

3.4 Conclusion and Future Work

In this paper, we presented a novel sensor datum for intrusion detection with the main aspect of a resilient cost model. The cost model pays respect to the hierarchical connection between several memory components and takes into account that the speed of each level slows down from top (first-level cache) to bottom (main memory). We also introduced two equations which are capable of calculating the overall *average memory access time* and the *interference ratio* with respect to demand and coherence related misses.

As implied by the model introduced in the previous chapters, we assume the observation will be triggered by a processing unit other than the central processing unit. We have decided to make use of "a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth" [12] in order to execute our observation algorithms. So, the next step will be taken by examining existing pattern matching algorithms with regard to their ability to be executed by a general purpose coprocessor. We will also start to develop our own algorithms with regard to a high degree of parallelism. Furthermore, our future work will contain a proof of concept implementation of the previous described observation method, which will include the implementation of a chosen algorithm, as well as a kernel module serving as an interface between the GPU and the main memory of the host.

Bibliography

[1] AGARWAL, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Ph.D. thesis, Stanford University, Stanford, CA, USA, Jan. 1987. 49

[2] ALBERS, S., FAVRHOLDT, L. M., AND GIEL, O. On Paging with Locality of Reference. *Journal of Computer and System Sciences 70*, 2 (Mar. 2005), 145–175. doi:10.1016/j.jcss.2004.08.002. 49

[3] ALMGREN, M., LINDQVIST, U., AND JONSSON, E. A Multi-Sensor Model to Improve Automated Attack Detection. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)* (Boston, MA, USA, Sept. 2008), R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., vol. 5230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 291–310. doi:10.1007/978-3-540-87403-4_16. 44

[4] CAVALLARO, L., SAXENA, P., AND SEKAR, R. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2008)* (Paris, France, July 2008), D. Zamboni, Ed., vol. 5137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–163. doi:10.1007/978-3-540-70542-0_8. 43, 53

[5] DENNING, P. J. The Working Set Model for Program Behavior. *Communications of the ACM 11*, 5 (May 1968), 323–333. doi:10.1145/363095.363141. 8, 46

[6] DENNING, P. J., AND SCHWARTZ, S. C. Properties of the Working-Set Model. *Communications of the ACM 15*, 3 (Mar. 1972), 191–198. doi:10.1145/361268.361281. 46

[7] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2006. 8, 18, 44, 46, 47, 48, 49

[8] MCEVOY, T. R., AND WOLTHUSEN, S. D. Concurrent Host Integrity Protection and Intrusion Detection. Unpublished early draft, 2009. 44, 45, 54, 57, 67, 91, 93

[9] MCEVOY, T. R., AND WOLTHUSEN, S. D. Host-Based Security Sensor Integrity in Multiprocessor Environments. In *Proceedings of the 6th Information Security Practice and Experience Conference (ISPEC 2010)* (Seoul, Korea, May 2010), J. Kwak, R. Deng, Y. Won, and G. Wang, Eds., vol. 6047 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 138–152. doi:10.1007/978-3-642-12827-1_11. 24, 43, 44, 53, 54, 57, 58, 59

[10] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Computer Society, pp. 421–430. doi:10.1109/ACSAC.2007.21. 10, 43, 53, 82

[11] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)* (Seattle, WA, USA, Mar. 2008), S. Eggers and J. Larus, Eds., ACM, pp. 308–318. doi:10.1145/1353535.1346321. 30, 43

[12] NVIDIA CORPORATION. NVIDIA CUDA Programming Guide 2.2.1, May 2009. (Last checked: August 22, 2012). 20, 50

[13] OPLINGER, J., AND LAM, M. S. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (San Jose, CA, USA, Oct. 2002), ACM, pp. 184–196. doi:10.1145/605397.605417. 43

[14] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[15] PETRONI, JR., N. L., FRASER, T., WALTERS, A., AND ARBAUGH, W. A. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium* (Vancouver, Canada, Aug. 2006), USENIX Association, pp. 289–304. (Last checked: August 22, 2012). 44

[16] PETRONI, JR., N. L., AND HICKS, M. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)* (Alexandria, VA, USA, Oct. 2007), ACM, pp. 103–115. doi:10.1145/1315245.1315260. 44

[17] TORNG, E. A Unified Analysis of Paging and Caching. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS 1995)* (Milwaukee, Wi, USA, Oct. 1995), IEEE Computer Society, pp. 194–203. doi:10.1109/SFCS.1995.492476. 49

[18] WILLIAMS, P. D., AND SPAFFORD, E. H. CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection. *Computer Networks 51*, 5 (Apr. 2007), 1284–1298. doi:10.1016/j.comnet.2006.09.011. 30, 43, 67, 93, 94

[19] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure Coprocessor-Based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, July 2002), ACM, pp. 239–242. doi:10.1145/1133373.1133423. 23, 24, 43, 53, 54, 59, 61, 62, 67

Chapter 4

Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection

Abstract

The high computational power of graphics processing units (GPU) is used for several purposes nowadays. Factoring integers, computing discrete logarithms, and pattern matching in network intrusion detection systems (IDS) are popular tasks in the field of information security where GPUs are used for acceleration. GPUs are commodity components and are widely available in computer systems which would make them an ideal platform for a wide-spread IDS.

We investigate the feasibility to use current GPUs for asynchronous host intrusion detection as proposed in a former work and come to the conclusion that several constraints of GPUs limit the use for concurrent and asynchronous off-CPU processing in host IDSs. GPUs have restrictions in terms of continuity, asynchronism, and unrestricted access to perform this task. We propose an observation mechanism and discuss current constraints on autonomous use of standard GPU components for intrusion detection. Finally, we come to the conclusion that several modifications to graphics cards are necessary to enable our approach.

4.1 Introduction

The detection and prevention of malware attacks at an early state is highly desirable. Steadily rising complexity of attacks makes it increasingly difficult to detect malware with commonly used techniques. Malware uses hiding techniques, obfuscation approaches like polymorphism or metamorphism, or obscure targeted attacks to bypass signature-based detection systems. A successful malware attack compromises the operating system's (OS) Trusted Computing Base (TCB) and, therefore, all components which base on it [29]. Rootkits, for example, undermine the integrity of the system and any installed host intrusion detection mechanisms. After an attack occurred, an alternative reconstructed system state is established. The new state is also legitimate and plausible for the system. Therefore, the system cannot detect its compromised state. Once suitable hiding techniques were used, the detection of these reconstructed states is problematic for traditional computational models [18, 8]. The states cannot be distinguished from legitimate states [16].

In a previous work [16], a defensive detection mechanism is described which is not bound to the traditional model of computation. The mechanism uses approaches from distributed systems [14] and relies on the use of an increasing amount of mostly idling multi-processor (multi-core) host systems to achieve a non-deterministic concurrency. A large number of deployed simultaneous observer sensors monitor attacks and defensive measures. The approach does not rely on obfuscation, concealment or insulation to prevent manipulation of the attacker. It benefits from non-deterministic concurrency which renders it impossible to predict and control the mechanism. Even if the attacker knows the mechanism, he cannot benefit from this knowledge because the attacker cannot control or predict the observation. Respectively, he would be observed at attacking the mechanism.

[15] proposes the integration of coprocessors or other non-standard sensors like graphics chipsets or PCI processing devices. With this approach, a self-protection of the IDS monitor is achieved since a compromise of the host does not affect the independent coprocessor [29]. This mechanism is generally immune to tampering and enables intrusion detection techniques such as sampling memory areas, process flow, and memory shadowing. For attackers this detection mechanism would be practically not discernible by running isolated from the system and, therefore, hidden from the attacker. Furthermore, the outsourcing of the intrusion detection to off-CPU components could prevent the performance impact on the host system almost entirely. The observation of the system does in fact require many threads, since the sensors are stateless and behave probabilistically. A large number of kernel threads would have a noticeable effect on the performance of the system.

Graphics cards with one or more GPUs are commodity components and are widely available in current computer systems. This would make them an ideal platform for a wide-spread IDS. Modern graphics cards are connected via a fast PCI Express 2.0 (PCIe) x16 interface and allow high performance parallel computations. However, constraints of current GPU's hardware and programming architecture render it difficult to use them as autonomous sensors for asynchronous memory observation. GPU program code cannot run independently of the initiating instance which resides on the host system. Thus, GPU kernels cannot assure continuity of examination in the case of a compromised host system.[1] Furthermore, APIs of current GPUs offer only a user space interface – GPU kernels cannot access kernel space memory due to limitations in the graphics driver. Therefore, no access to the host's memory for observing sensitive OS structures is possible.

In our work we outline the occurring constraints on autonomous use of current GPUs for asynchronous observations and intrusion detection and present an observation mechanism which we will discuss afterwards. The design goals of the proposed independent detection approach contain the unfeasibility for an attacker to predict, attack, or control the IDS mechanism [16].

The remainder of this paper is organized as follows: In Section 4.2, we investigate on asynchronous memory access and then, in Section 4.3, we review GPUs, especially the architecture of the control-flow and the memory architecture. In Section 4.4, we propose our observation mechanism, followed by Section 4.5 where we evaluate GPUs as independent auditors. Then, in Section 4.6, we discuss the occurring constraints, propose modifications to GPUs, and give an overview of most recent related work in Section 4.7. Finally, in Section 4.8, we draw our conclusions and present challenges concerning future work.

4.2 Asynchronous Memory Access

Modern computer's peripheral devices use direct memory access (DMA) to access system memory independently of the CPU. In most cases, the system's chipset offers two DMA controllers with several independent DMA channels for this task. DMA controllers allow concurrent data transfers providing one transfer per channel. Therefore, DMA is another source of parallelization [1].

In a common control-flow, the CPU initiates the transfer, continues with normal operation while the transfer is in progress, and gets notified from the DMA controller when the operation has been done. Similarly, a DMA engine in a graphics card can transfer data while the processing element carries on with its own task. The graphics card typically uses bus mastering DMA to let the device take control of the PCIe bus and perform the transfer itself. Especially in stream processing DMA plays a vital role since it is essential to have data processing and transfers in parallel in order to achieve sufficient processing throughput. Nevertheless, in such an environment a great number of threads have to share few DMA channels.

[1] Presupposed the application of host-side programming and runtime frameworks.

It is important to say that memory accesses are generally expensive [26]. Even at read only accesses they reduce the possible concurrency and force synchronizations between the units in the worst case. In addition, memory accesses via DMA can lead to a cache coherency problem which will affect the overall system performance even though they are bypassing the CPU and, therefore, do not affect the CPU caches directly.

4.3 GPU Architecture

General-purpose computing on graphics processing units (GPGPU) allows the use of stream processors for highly parallel non-graphics calculations. GPUs are based on a stream processing architecture and follow a data-centric model. They have a SIMD (Single Instruction, Multiple Data) architecture which allows achieving data parallelism. Each *stream processor* performs the same *kernel* instructions in parallel on different pieces of a distributed data stream [11].

The GPU acts as a coprocessor of the CPU. Data-parallel code is outsourced in designated functions which are then executed on the GPU. With asynchronous function calls, the CPU continues to execute instructions while the GPU is running a kernel.

The performance of GPUs benefits from cost intensive operations with few I/O and global memory access bandwidth. Furthermore, it profits from a high amount of data-parallel operations which process an input data stream simultaneously. By executing multiple parallel copies of program code, the GPU can utilize a large number of the available arithmetic logical units (ALU), thus reaching a high load factor. In addition, the latency of dynamic random access memory (DRAM) read operations can be hidden by utilizing thousands of simultaneously running threads. Also, a high data locality, whereupon data is produced once, immediately processed, and never read again is beneficial [6].

4.3.1 Memory Architecture

The memory architecture of GPUs consists of three major memory domains: host (CPU) memory, PCIe memory, and local (GPU) memory. Host memory cannot be accessed by a GPU kernel; only code running on the host can access it. PCIe memory is a part of host memory which is reserved for PCIe usage. It can be both modified from host programs and GPU. Local memory is only accessible by the GPU and cannot be accessed through the CPU [1]. There are different local memory types available which differ only in caching algorithms and access models.

Basic DMA memory transfers from host to GPU involve two memory copy steps: between host and PCIe, and between PCIe and GPU's local memory. Since DMA transfers require page-locking, memory regions have to be copied to page-locked memory areas managed by the device driver. After that they are transferred to GPU's local memory. With proper transfer management and the use of system page-locked memory (host memory remapped to the PCIe memory space) copying between host memory and PCIe memory can be skipped [1]. However, this memory access is limited to user space virtual memory of the user process which called the coprocessor. Accessing of other processes' virtual memory is not possible. Whereas, benefit of this design is that the GPU cannot accidentally or maliciously corrupt the memory content of another process [10], it limits the GPU's ability to scan host's kernel space memory.

In a multi-GPU environment one GPU cannot access another GPU's memory directly. Transfers are made via the PCIe bus.

4.3.2 Control-Flow

The host application does not interact with the GPU directly. A driver layer translates and issues commands to the hardware on behalf of the application. Most commands to the GPU are buffered in a command queue on the host-side. The command queue is flushed to

the GPU, and the commands are processed by it, only when a kernel program is executed. Flushing sends the current state of the command queue to the GPU. Transfers from the host to the GPU are done either by the command processor or by the DMA engine. The communication is done via the PCIe bus [1].

Before calls can be issued to the GPU, a CPU-GPU context has to be established [4, pp. 23-24]. In terms of NVIDIA, such a context is established by the first CUDA call that changes the state. State changes occur, for example, by calling `cudaMalloc` or after a kernel launch. A context is destroyed by an explicit call of an appropriate function or after the host thread terminates. Additionally, the CUDA resources are allocated per context.

One context can be maintained by a host thread at a time. In a multi-GPU environment every GPU needs a host thread. Additionally, multiple host threads can establish contexts with the same GPU. To enable that, the driver handles time-sharing and resource partitioning for the GPUs.

To access a host's main memory from a GPU kernel the memory has to be copied to the graphics card's device memory. GPUs use a DMA architecture to transfer data between the device memory and the host memory. DMA requires page-locking of the host's memory for asynchronous `memcpy`. Pageable memory cannot be copied asynchronously since the OS may move it or swap it out to disk before the GPU has finished using it [20]. The graphics card driver maintains page-locked memory for this purpose where the requested memory is copied before the actual DMA transfer via the PCIe bus. Alternatively, current GPUs allow mapping page-locked host memory into the address space of the GPU.

GPUs can execute instructions in parallel, separately from each other. For this task they can maintain several thousand simultaneously running lightweight threads. With the simultaneous execution of those threads they solve the problem of high memory access latencies. If a DRAM access occurs, the thread which is waiting for data from memory is suspended and another thread can be executed without latencies (latency hiding). Compared to costs of hundreds of cycles per thread change for CPUs, GPUs can switch several threads per cycle.

The two mentioned mechanisms, DMA and the simultaneous execution of threads, allow the GPU to execute parallel memory accesses.

There are certain bottlenecks in the GPU architecture in terms of interaction and data flow. For this reason, the GPU program often relies on the CPU to manage actions such as control-flow, process termination, and memory I/O accesses. The CPU-GPU link has relatively low performance and high latencies compared to other accesses of GPUs. As a result, the GPU cannot take advantage of the performance gain due to its parallel execution of kernels.

Copies between host memory and PCIe memory usually are in the hundreds of MB/s. Copies between the PCIe memory and the GPU memory reach a theoretical (actual transfer rates are CPU and chipset dependent) throughput of 4 GB/s for PCIe x16 Generation 1 and 8 GB/s for Generation 2 in each direction. (Local) memory bandwidth is in the tens to hundred GB/s range due to the usage of several memory controllers. Nevertheless, it suffers from very high latencies (several hundred cycles) and is not cacheable.

4.3.3 Programming Interfaces

Currently the GPU market consists of two major vendors: NVIDIA with its CUDA SDK [19] and its direct competitor AMD/ATI with its Stream SDK [2]. Both participate in the upcoming open standard OpenCL [12] which aims at interoperability of GPUs with other computing environments.

Unfortunately, GPU hardware is proprietary. With the exception of AMD/ATI GPUs, for which partially open documentation is available [3], there is neither open code nor any specification available for the underlying kernel interfaces of the above mentioned technologies. For example, NVIDIA's hardware documentation is a closely guarded trade

secret. This renders it very difficult to implement functionality on the GPU without using available APIs.

Several programming APIs, e.g., NVIDIA's CUDA Driver API or Runtime API are available for developers. In its functionality, GPU programming APIs are very similar to the C language. But conceptionally they offer, for example, no support for timers and no concurrency between jobs. Moreover, it is not possible to observe data at different levels of abstraction [16] because graphics cards cannot access OS APIs.

4.4 Observation Mechanism

The intrusion detection mechanism [16] mentioned in the introduction relies on non-deterministic concurrency. With this approach the mechanism detects violations by the linearization of observations of invariants and relations among critical components of the OS, so-called *semantic chokepoints*. Semantic chokepoints describe data structures in the system which the malware must manipulate in order to remain undetected. Thus, the precise functions of the malware need not be known. The preservation of the integrity of these few points already helps to detect a number of attacks.

The points have to be monitored simultaneously by sensors from different and independent points of view at frequent, but probabilistically determined intervals [15]. In [16] this is accomplished by using processes residing on different CPUs, viewing higher and lower level APIs within the kernel structure. Additionally, direct measurement of selected areas in memory to provide a concurrent four-dimensional view of system activity is mentioned.

Our proposed autonomous observation mechanism consists of two components: one or many concurrent running GPU kernels and a module on the host system which consists of a device driver and the main application. The latter defines the tasks regarding the observation, assigns the required number of GPU kernel threads for each observation, and initiates the observation mechanism on the GPU. Once initiated, the GPU repeats the execution of the observation at predefined intervals. The measurement frequency is a tunable parameter and defines the granularity of observation and affects the performance of the host system.

The observations are weakly synchronized [16]. After every measurement, the sensors exchange messages with each other to compare results and report consistence of the state. Therefore, if using a multi-GPU environment, communication between the GPUs has to be taken into account.

If using multiple GPUs, kernels can simultaneously observe the host system. Thus, independent points of view can be realized within this concurrent multi-GPU environment. Our approach uses at least one graphics card which can reside integrated on the mainboard or on a PCIe expansion card.

Using GPUs, it is not possible to perform cross-sectional analysis of OS kernel layers and of relations between the current states of kernel structures. This refers to the off-OS operation of the GPU sensors. Nevertheless, every sensor has direct access to selected areas of the host system's main memory. Therefore, GPUs can monitor these areas at frequent intervals.

Our observation mechanism profits from asynchronous and parallel measurements at detecting inconsistencies in the system state. In particular, the comparison of observations makes it possible to create linearizations which show transitions occurring in the system state. Thus, the breakdown of known relations indicates behavioral inconsistencies.

Furthermore, heuristics which are implemented on the GPU can reconstruct states from asynchronous observations of kernel state by treating inconsistencies in state as transitions and then comparing the set of all possible transitions from the model with the set of observed transitions to uncover any unfeasible sequencing in states [16]. Thus, the integrity of important kernel structures, for example, process structures or function tables can be

monitored. Beside these heuristically approaches, fast pattern matching with signatures of malicious code can be implemented with the proposed observation mechanism.

The GPU kernels subsequently write their observation results to a secured memory area accessible by the kernel module. The kernel module then creates appropriate alerts.

With periodic tests of the integrity of the mechanism, multiple sensors in a multi-GPU environment can protect the observation mechanism by observing data elements associated with the observation mechanism itself [16].

Last but not least, as for all security applications the system has to be started in a trusted environment, e.g., measured by a trusted platform module (TPM). In this case the intrusion detection mechanism is required to load as part of the boot process of the host system before the OS gets started. Once started, the observation mechanism should operate throughout the operation of the system.

4.5 GPUs as Independent Auditors

In the following section, we investigate on the limitations of current GPUs acting as co-processors for intrusion detection. We evaluate the proposed autonomous GPU approach with the properties of a former PCI add-in card solution [17, 24]. These papers define a set of properties which have to be accomplished in order to indicate that machine A is an out-of-band or independent auditor of machine B. In our case machine A is a GPU (our coprocessor) and machine B is our host system. The following list illustrates the requirements of an independent auditor. This list covers the most important requirements and is not intended to be exhaustive.

- *Unrestricted access: Machine A must have unrestricted access to the internal devices of machine B to be verified or needed for the verification, including peripherals, hard disks and interrupts.*
  The intrusion detection mechanism running on the GPU does not have unrestricted access to the physical memory and other components of the host system as the controlling host application is running in the user address space. Only a subset of the full range of the host memory can be accessed by the GPU. This concludes that the GPU does not have unrestricted access to the host system's resources.

- *Secure transactions: The channel used by the independent auditor to retrieve the data should be a secure channel, meaning a channel which cannot be eavesdropped or intercepted, nor modified.*
  The channel is not secure. Once the graphics card has control of the PCIe bus commands and data passing through host-side controllers may be eavesdropped or intercepted (cf. [25]).

- *Inaccessibility: Machine B must not have access in any way to the internal components of machine A, including memory and internal interrupts.*
  The host system has no access to the internals of the GPU. Access is only possible indirectly using the GPU's API routines.

- *Continuity: Machine A must run immediately after machine B has setup the internal devices and is in a known trusted state. From this moment, Machine A must run continuously, independently of the behavior of machine B, particularly when it has been compromised.*
  Code in the GPU cannot be started before the host system booted the OS. The OS can be in a not trusted state at that moment. Moreover, the GPU is not independent of the host system.

- *Transparency: To the maximum degree possible, machine A should not be visible to the host processor. At a minimum, it should not disrupt the host's normal activities and should require no changes to the host's OS or system software.*

Transparency would be possible only if we had unrestricted access to the host system. Access to the host system memory would be possible transparently using DMA, if somehow unrestricted access can be achieved.

- *Verifiable software: All the code running in machine A must be trusted and verifiable. This, at least, implies that all running software in machine A must have the source code available. This includes the firmware, OS and user space programs in machine A.*
  GPUs are not verifiable. There is no source code available for the firmware.

- *Non-volatile memory: Machine A must be capable of retaining a record of the alerts even in the event of a power failure or reboot. Hence, machine A should have some non-volatile storage to record sensitive data.*
  There is no non-volatile memory available on current GPUs to retain record of alerts. Also GPUs do not have direct access to disks or other storage media.

- *Physically secure: Machine A should be physically secure.*
  GPUs are built-in into the host system and, therefore, can be treated as physically secure.

- *Sufficient processing power: The coprocessor will, at a minimum, need to be able to process large amounts of memory efficiently.*
  A GPU can handle large amounts of memory efficiently due to its data-parallel processing architecture.

- *Out-of-band reporting: Machine A must be able to securely report the state of machine B. To do so, there must be no reliance on a possibly compromised host, even to perform basic disk or network operations. Machine A must have its own secure channel to the admin station.*
  GPUs do not have appropriate external communication interfaces.

In addition to the constraints mentioned above, a major disadvantage of using autonomous coprocessors in intrusion detection is that they are unable to interpose themselves between the malicious process and the OS to collect IDS data and analyze the attacks. To examine and additionally modify the host's state the coprocessor has to share interfaces with the host processor. The number and choice of interfaces determines the degree to which intrusion detection is possible [29]. Furthermore, because coprocessors operate asynchronously they suffer limitations in their ability to reconstruct kernel states [16].

Current GPU APIs offer only user space interfaces. Thus, GPU kernels can only be initiated via a user mode process.

With current GPUs, independent execution of program logic is not possible because of the host-bound kernel execution. Once the host's user mode process which started the kernel exits, the running GPU kernel gets terminated. In terms of malware infection this means that if the host is subverted the GPU kernel cannot run in a trusted state anymore. In addition, implementations in user mode or kernel drivers are not asynchronous because they are dependent on the OS's scheduler. For this reason, asynchronous measurement is not possible.

The GPU kernel has no unrestricted access to the host's physical memory so it cannot directly interact with the host system and directly scanning for malware in host's memory is not feasible. Host memory has to be available in the graphics card's address space before the GPU kernel can access it. This can be done either by explicitly copying the memory to the graphics card or by mapping memory to pre-allocated page-locked user space memory areas which can be accessed by the GPU via a zero-copy mechanism [20].

The observation of the memory is based on sampling. Thus, there is no guarantee to detect the attack in time. An appropriate combination of placement of the samples, the number of samples, the sampling period, and the period distribution has to be found [29].

The number of samples is limited by the data access latency of the graphics card. Additionally, in a multi-GPU environment the data exchange between GPUs via the PCIe bus is a limiting factor.

Another limitation refers to the maximum runtime of a kernel. Current GPUs are not built to run graphics calculations and time intensive or infinite general purpose computations simultaneously. For the current Microsoft Windows OSs the window manager terminates any kernel externally via the driver that runs more than a certain period of time (approximately five seconds). This mechanism exists to prevent system hangs where the display will freeze until the kernel has finished executing.

4.6 Discussion

The autonomous use of standard GPU components without modifications of their architecture is not feasible due to components which control the GPU from host-side. For example, the graphics card driver makes the approach dependent on the OS's scheduler. Moreover, necessary hardware components are missing which would allow using graphics cards as coprocessors for intrusion detection.

4.6.1 Limitations

Working around the GPU APIs is not possible without further investigation of the closed architecture of the involved driver and the graphics card. There is currently no way to start kernels from within a kernel driver. [5] uses a microdriver architecture to circumvent this limitation. Although technically feasible it is unlikely that GPU vendors will provide kernel interfaces which directly access GPUs in the near future [5]. This is probably caused by stability issues which would arise. If the kernels are initiated from user address space, malware which resides in kernel space can interfere the execution with minimum effort.

Another limitation regards to the maximum GPU kernel runtime. To prevent system hangs, graphics calculations are limited in their maximum execution time. This prevents freezing screens if a kernel computation fails and lasts too long. Although this behavior can be changed by increasing the watchdog's time threshold or by completely disabling the watchdog, these modifications are recommended only for debugging and testing purposes[2] and are not feasible in a real environment.

A potential solution for this problem is to use NVIDIA's new GPU generation [21] which supports simultaneous execution of multiple kernels. This way, displaying tasks can stick to short time limits and computational tasks can have no runtime limitations. Alternatively, the kernel execution can be broken into smaller bits. However, this is a major constraint for the design of the intrusion detection algorithm.

Another solution is to use a dedicated card for kernel computations which has no display connected. But the demand for an additional graphics card for GPU processing only breaks the benefit of using commodity hardware for intrusion detection. It is a requirement that apart from the intrusion detection the GPU should be usable for displaying tasks. Only then our proposed wide-spread IDS approach is practicable.

4.6.2 Architectural Modifications

To use GPUs as independent auditors we propose several modifications which have to be made regarding their architecture. First of all, a mechanism needs to be devised to get unrestricted access to the host's physical memory. A potential solution is using unrestricted DMA as previously done in [24]. The current DMA functionality is too restrictive and does not allow the required accesses.

[2] http://msdn.microsoft.com/en-us/library/ms797877.aspx

In terms of continuity, the operation of the IDS on the GPU has to be started at an early stage of the host system's boot process. As there is no driver loaded at that time, the GPU is independent of the OS. To support such a functionality, a new independent operation mode has to be implemented which still works with the normal graphics driver activity. In addition, this approach circumvents the GPU APIs with its CPU-GPU context behavior which terminates the GPU kernel at certain events. The GPU kernel must run independently of the host's software routines and especially must not have a maximum runtime limitation.

Additionally, hardware vendors have to open their code and/or their architecture for at least the sensitive sections which affect the secure operation of the IDS.

Moreover, current GPUs lack of a non-volatile memory to store records of alerts. The only permanent storage is an EEPROM which holds the graphics card's BIOS code. By providing a small amount of flash memory on the graphics card – accessible by the GPU kernel – this limitation can be resolved.

4.6.3 Efficiency

Unfortunately, we cannot compare our proposed observation mechanism with existing IDS schemes due to the current unfeasibility of creating a proof of concept.

In general, GPU's parallelization capabilities should be beneficial compared to existing solutions based on common PCI devices. Especially at pattern matching much higher performance can be expected than from other IDSs.

Because the CPU is not directly involved into the IDS, only few CPU cycles are occupied. Due to most of the time underutilized GPU kernels especially in desktop environments, only minor decreases on graphics performance can be expected. Nevertheless, the asynchronous memory accesses via DMA can lead to a cache coherency problem which will affect the overall system performance.

Additionally, with current possibilities it is not feasible to gain a complete picture of the system's host memory. As proposed, major modifications of GPU's architecture are necessary to enable this functionality.

The actual advantage of our proposed observation mechanism is the fast connection to the mainboard's chipset and the high computational throughput reached through parallelization of the GPU kernels. In addition, computation performance increases rapidly with every new GPU revision.

Furthermore, an increasingly amount of computers have access to GPU's computation resource. This wide-spread availability makes the idea of an IDS worth investigating. Moreover, keeping in mind that integrated GPU solutions in mobile computers and devices already use parts of host's main memory our mechanism shows its strength regarding the performance of memory observation.

4.6.4 Security

Since the proposed intrusion detection mechanism does not work independently, a compromise of the host system does affect its operation. Moreover, the DMA mechanism of the GPU has only read access to protect the integrity of the OS in case of a malfunction. Furthermore, processes on the host system cannot access GPU's resources directly – especially the local memory – since the graphics driver manages and restricts all interactions.

With current GPUs, it is not possible to secure the communication between the host components and the graphics card. As shown with a PCI-based cryptographic coprocessor [29], the communication with the IDS can be authenticated. Thus, only the main application is permitted to configure the IDS during its operation.

Regarding denial of service attacks or other compromises against the graphics driver, it has to be mentioned that driver isolation is a common topic [28, 7, 13, 27] and helps to

secure both the graphics card driver and the kernel driver used for the observation mechanism.

4.7 Related Work

Related work regarding independent auditors was proposed in [24] with a coprocessor-based kernel integrity monitor. The proposed monitor does not depend on the underlying OS. Based on a single-board computer which is embedded on a PCI add-in card the monitor is able to examine the host's RAM without the knowledge or intervention of the host kernel. Unrestricted memory access is gathered via DMA. Due to its architecture, any access to PCI memory corresponds directly to an access in the 32-bit physical address space of the host system [24]. Unfortunately, Rutkowska [25] showed that by manipulating certain chipset registers the memory acquisition via DMA can be, for example, redirected to garbage data.

Zhang et al. [29] used a PCI-based cryptographic coprocessor to implement host-based intrusion detection tasks on a tamper-resistant computing device which is designed to perform critical tasks in an environment in which physical attacks are possible. With this approach, messages from the sensors can be authenticated in contrast to our GPU approach.

Other solutions proposed virtual machine monitors [9, 23] to get a secured and complete picture of a host system's activity.

Although these mentioned systems offer relatively good observation platforms, they all use special hardware components which have no wide-spread availability compared to our standard GPU components approach. This renders their usage difficult in terms of host intrusion detection.

The necessity of an isolated graphics system was mentioned by Okhravi and Nicol [22]. They describe the design and implementation of a trusted graphics subsystem for high assurance systems. Nevertheless, their approach deals with problems at a higher abstraction level.

Seeger and Wolthusen [26] proposed a computational model and observation mechanism for the case of tightly coupled asymmetric concurrent processing units. They deal with costs of forced synchronizations and resource contention caused by observations.

4.8 Conclusion

In this paper we investigated the constraints of using GPUs as autonomous coprocessors for asynchronous observations and intrusion detection. GPUs with general purpose computing capabilities are wide-spread commodity hardware and deliver a fast coprocessor. They have a fast connection to the motherboard's chipset and have highly parallel computation units. Furthermore, graphics cards are a mostly underutilized computing resource and operate outside the OS. Under ideal conditions the intrusion detection mechanism is undiscoverable. As a result, a high attack resistance of the IDS monitor is achieved since a compromise of the host does not affect the coprocessor. Nevertheless, in comparison to the architecture of each CPU, the GPU's SIMD architecture offers only a very limited programming model. Memory accesses reduce the possible concurrency and workloads have to meet certain requirements in order for the GPU to perform best.

We presented an observation mechanism and showed that current GPUs have several major constraints in executing our proposed task. The list of requirements of independent auditors showed that several basic requirements are not fulfilled. GPU kernels cannot run independently from the host system and have no unrestricted access to the physical memory of the host system. In addition, graphics cards have no appropriate permanent storage mechanism.

Since our approach is not feasible with current graphics hardware, we suggested several modifications to graphics cards which are necessary to enable our proposed mechanism.

In our future work, we will investigate on modifying the existing graphics device interfaces to reach our goal: the usage of fast commodity hardware for asynchronous host IDSs. Additionally, we want to focus on appropriate algorithms to process our observations and we want to measure how the generated asynchronous memory accesses affect the performance of the host system.

Acknowledgment

The authors would like to thank Eckehard Hermann, Christoph Kemetmüller, Julian Knauer, Sufiyan Siddique, and Dieter Vymazal for their insightful comments.

Bibliography

[1] ADVANCED MICRO DEVICES. ATI Stream Computing – Technical Overview, 2009. (Last checked: August 22, 2012). 54, 55, 56

[2] ADVANCED MICRO DEVICES. ATI Stream SDK, 2010. (Last checked: August 22, 2012). 56

[3] ADVANCED MICRO DEVICES. Open GPU Documentation, 2010. (Last checked: August 22, 2012]). 56

[4] BRADLEY, T. Advanced CUDA Optimization – 3. Execution, 2010. (Last checked: August 22, 2012). 56

[5] BRINKMANN, A., AND ESCHWEILER, D. A Microdriver Architecture for Error Correcting Codes Inside the Linux Kernel. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC 2009)* (Portland, OR, USA, 2009), ACM, pp. 1–10. doi:10.1145/1654059.1654095. 60

[6] BUCK, I., AND HANRAHAN, P. Data Parallel Computation on Graphics Hardware. Tech. rep., Stanford University, 2003. (Last checked: August 22, 2012). 55

[7] BUTT, S., GANAPATHY, V., SWIFT, M. M., AND CHANG, C.-C. Protecting Commodity Operating System Kernels from Vulnerable Device Drivers. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)* (Honolulu, HI, USA, Dec. 2009), IEEE Computer Society, pp. 301–310. doi:10.1109/ACSAC.2009.35. 61

[8] CAVALLARO, L., SAXENA, P., AND SEKAR, R. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2008)* (Paris, France, July 2008), D. Zamboni, Ed., vol. 5137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–163. doi:10.1007/978-3-540-70542-0_8. 43, 53

[9] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed Systems Security Symposium (NDSS 2003)* (San Diego, CA, USA, Feb. 2003), Internet Society, pp. 191–206. (Last checked: August 22, 2012). 62

[10] GELADO, I., KELM, J. H., RYOO, S., LUMETTA, S. S., NAVARRO, N., AND MEI W. HWU, W. CUBA: An Architecture for Efficient CPU/Co-processor Data Communication. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS 2008)* (Island of Kos, Greece, June 2008), ACM, pp. 299–308. doi:10.1145/1375527.1375571. 55

[11] HILLIS, W. D., AND STEELE JR., G. L. Data Parallel Algorithms. *Communications of the ACM 29*, 12 (Dec. 1986), 1170–1183. doi:10.1145/7902.7903. 55

[12] KHRONOS GROUP. OpenCL, 2010. (Last checked: August 22, 2012). 56

[13] LEONTIE, E., BLOOM, G., NARAHARI, B., SIMHA, R., AND ZAMBRENO, J. Hardware-enforced Fine-grained Isolation of Untrusted Code. In *Proceedings of the 1st ACM Workshop on Secure Execution of Untrusted Code (SecuCode 2009)* (Chicago, IL, USA, Nov. 2009), ACM, pp. 11–18. doi:10.1145/1655077.1655082. 61

[14] MCEVOY, T. R., AND WOLTHUSEN, S. D. Using Observations of Invariant Behavior to Detect Malicious Agency in Distributed Environments. In *Proceedings of IT Incident Management and IT Forensics (IMF 2008)* (Mannheim, Germany, Sept. 2008), vol. 140 of *Lecture Notes in Informatics*, GI, pp. 55–72. (Last checked: August 22, 2012). 53, 103

[15] MCEVOY, T. R., AND WOLTHUSEN, S. D. Concurrent Host Integrity Protection and Intrusion Detection. Unpublished early draft, 2009. 44, 45, 54, 57, 67, 91, 93

[16] MCEVOY, T. R., AND WOLTHUSEN, S. D. Host-Based Security Sensor Integrity in Multiprocessor Environments. In *Proceedings of the 6th Information Security Practice and Experience Conference (ISPEC 2010)* (Seoul, Korea, May 2010), J. Kwak, R. Deng, Y. Won, and G. Wang, Eds., vol. 6047 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 138–152. doi:10.1007/978-3-642-12827-1_11. 24, 43, 44, 53, 54, 57, 58, 59

[17] MOLINA, J., AND ARBAUGH, W. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security (ICICS 2002)* (Singapore, Dec. 2002), R. Deng, F. Bao, J. Zhou, and S. Qing, Eds., vol. 2513 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 291–302. doi:10.1007/3-540-36159-6_25. 6, 8, 23, 24, 25, 58

[18] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Computer Society, pp. 421–430. doi:10.1109/ACSAC.2007.21. 10, 43, 53, 82

[19] NVIDIA CORPORATION. NVIDIA CUDA SDK. (Last checked: August 22, 2012). 56

[20] NVIDIA CORPORATION. CUDA 2.2 Pinned Memory APIs, 2009. (Last checked: August 22, 2012). 56, 59

[21] NVIDIA CORPORATION. Fermi – NVIDIA's Next Generation CUDA Compute Architecture, 2009. (Last checked: August 22, 2012). 60

[22] OKHRAVI, H., AND NICOL, D. M. TrustGraph: Trusted Graphics Subsystem for High Assurance Systems. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)* (Honolulu, HI, USA, Dec. 2009), IEEE Computer Society, pp. 254–265. doi:10.1109/ACSAC.2009.31. 62

[23] PAYNE, B. D., AND LEE, W. Secure and Flexible Monitoring of Virtual Machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Computer Society, pp. 385–397. doi:10.1109/ACSAC.2007.10. 62

[24] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[25] RUTKOWSKA, J. Beyond The CPU: Defeating Hardware Based RAM Acquisition, Feb. 2007. (Last checked: August 22, 2012). 58, 62, 69, 95

[26] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[27] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (Bolton Landing, NY, USA, Oct. 2003), ACM, pp. 207–222. doi:10.1145/945445.945466. 61

[28] WITCHEL, E., RHEE, J., AND ASANOVIĆ, K. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)* (Brighton, UK, Oct. 2005), vol. 39, ACM, pp. 31–44. doi:10.1145/1095809.1095814. 61

[29] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure Coprocessor-Based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, July 2002), ACM, pp. 239–242. doi:10.1145/1133373.1133423. 23, 24, 43, 53, 54, 59, 61, 62, 67

Chapter 5

The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation

Abstract

Intrusion detection relies on the ability to obtain reliable and trustworthy measurements, while adversaries will inevitably target such monitoring and security systems to prevent their detection. This has led to a number of proposals for using coprocessors as protected monitoring instances. However, such coprocessors suffer from two problems, namely the ability to perform measurements without relying on the host system and the speed at which such measurements can be performed.

The availability of smart, high performance subsystems in commodity computer systems such as graphics processing units (GPU) strongly motivates an investigation into novel ways of achieving the twin objectives of self-protected observation and monitoring systems and sufficient measurement frequency. This, however, gives rise to performance penalties imposed by memory synchronization particularly in non-uniform memory architectures (NUMA) even for the case of direct memory access (DMA) transfers.

Based on prior work detailing a cost model for synchronizations of memory access in such advanced architectures, we report an experimental validation of the cost model using an IEEE 1394 DMA bus mastering environment, which provides full access to the measurement target's main memory and involves multiple bus bridges and concomitant synchronization mechanisms. We observed more than 25% performance degradation, highlighting the need for efficient sampling strategies for both memory size and a preference for quiescent data structures for monitoring executed by off-host devices.

5.1 Introduction

Despite the fact that the use of coprocessors for host intrusion detection (ID) has been proposed years ago (cf. [19, 18]) they are currently not used in this domain. Even though their practical applicability has been shown by [12][1], commercial host ID software is still being installed on the host and executed by its CPUs.

In contrast to this, coprocessors are applied in the context of network ID (cf. [17, 4]). The ease of accessing data from network traffic in contrast to data on a host system (i.e., the host's main memory) and putting it under the audit of an auxiliary processor has contributed its part to this development.

The use of graphics chipsets or PCI processing devices (cf. [10]) and GPUs in general (cf. [16]) for host intrusion detection has been mentioned in previous work. While [10] mostly deal with self-protection of the observing component itself, [16] introduce a model capable of expressing the *costs* such observations can cause. That is, the authors pay special attention to concurrent memory accesses, particularly in a NUMA architecture as represented

[1] This included a proof of concept implementation called CoPilot.

by a standard multi-core (and especially multi-processor) system. The actual feasibility of a GPU in order to perform host intrusion detection was the subject of [14].

Although the model proposed in [16] and utilized here is intended mainly for high-speed interconnections such as these offered by GPUs and similar components, it is applicable to all non-uniform concurrent memory access architectures. The read-only (observation) access to a state variable of a target process can cross several cascading memory hierarchy layers with the ultimate shared resource in such architectures generally being main memory. As snooping and cache coherence protocols allow the efficient but forced synchronisation without the possibility of intervention on the part of software components, this provides a mechanism for concurrent state observations that cannot be corrupted or compromised. While [16] is concerned about observations executed by the GPU, similar effects with regard to resource contention and ultimately performance degradation on the host system can be caused by every memory access using DMA. Having in mind the limitations revealed by [14] we thus propose another hardware-based approach in this paper: That is, we exploited the ability of the IEEE 1394 technology in order to gain full access to the main memory of a connected computer. In order to make the observation effects as clear as possible, a workload generator produced alterations in a data structure of a predefined size. As we measured the CPU cycles consumed by our generator in both cases – with and without observation – we were able to quantify the loss of performance caused by observations executed by a processor other than the CPU of the host system.

The results obtained are of prime importance for the design of smart host intrusion detection algorithms running on off-host components. They provide central information with regard to the correlation of factors such as observation frequency, size of the data structure to be observed and the resulting performance degradation.

The remainder of this paper is structured as follows: In Section 5.2, we give an overview of the IEEE 1394 serial bus interface and point out the reasons for its applicability in our context. We also provide a historical insight into the main contributions regarding the field of physical attacks using the IEEE 1394 technology. We then present a description of our experimental setup that was used to obtain our results in Section 5.3 followed by a presentation of the actual results in Section 5.4. The paper is closed with a conclusion in Section 5.5 and a brief description of our future work in Section 5.6.

5.2 IEEE 1394

The IEEE 1394 interface, better known as FireWire, was initially developed by Apple Computer, Inc. in the 1980's [11]. Standardized by the Institute of Electrical and Electronics Engineers (IEEE) for the first time in 1995 (IEEE 1394-1995) [6], the latest changes have been published in October 2008 [7]. Due to its high speed and low overhead, today FireWire technology is mainly used for fast file transfer between external periphery such as mass data storages (e.g., hard drives, secure digital (SD) memory cards, etc.) and a computer. In contrast to the universal serial bus (USB), the FireWire standard allows for the communication between devices itself. Areas of application are here the communication between a digital camera and a printer, for instance.

In January 2000, version 1.1 of the open host controller interface (OHCI) specification was released by contributors from seven well-known computer companies, headed by Apple Computer, Inc. [1]. This interface is an implementation of the link layer protocol of the 1394 serial bus and empowers a broader variety of devices to take advantage of the IEEE 1394 interface. Furthermore, it is the OHCI that has features such as memory-mapping implemented in hardware which allows for a communication between two FireWire devices without the interaction of the operating system (OS) of either of them. Chapter 12 of the OHCI specification defines: *"When a block or quadlet read request or a block or quadlet write request is received, the 1394 Open HCI chip handles the operation automatically without involving software if the offset address in the request packet header meets a specific set of criteria..."* [1]. For

us, the relevant criterion to meet is the one that specifies that the address has to fall within the physical range of the target memory space. This range is defined by lower and upper bounds. While the offset is at address `48'h0` the address of the upper bound is either at `48'h000_FFFF_FFFF` or stored in the field `physUpperBoundOffset`[2]. This of course implies a security risk not only in theory but in practice as well, as we will see now.

5.2.1 Physical Attacks and Observations Using IEEE 1394

The properties of the IEEE 1394 technology noted above are also attractive in the security context for both attacks and forensic purposes. Although not within the scope of the present paper, we note that this use has ,e.g., been documented informally during the MacHack 17 convention in 2002, where a proof of concept for overwriting screen memory between two Apple Macintosh computers was demonstrated [13]. This was further elaborated by Dornseif et al. in 2004 using an Apple iPod for reading and writing to arbitrary memory locations in host systems without interacting with the target host operating system [5].

As expected and demonstrated by Boileau (cf. [2]), this is fully independent of any host operating system, although the device class must be known to the host system. This can, however, be easily emulated by setting the appropriate status registers to a known device and class, e.g., a mass storage device. Since all tools necessary for such an attack are freely available, having physical access to an enabled FireWire port on a computer is equal to having full access to its main memory. This includes use cases like changing the password protection code stored in main memory of a screen-locked computer in such a way that it accepts just any input. The intended lack of any access control mechanism or authentication schema between the communicating devices makes this possible.

For the purposes of attacks, performance considerations are largely irrelevant. However, forensic applications will aim to maximize speed. Particularly for the case of forensic memory capture, the fact that IEEE 1394 devices are typically coupled to main memory by one or more bus bridges makes this susceptible to chip-set modifications as, e.g., proposed and demonstrated by Rutkowska [15].

We note that the transitioning over multiple bus bridges is also affecting the measurements observed in the present paper as this implies not only matching different memory, bus, and device speeds, but also the use of multiple cache coherence and synchronization protocols.

5.3 Experimental Setup

In this section, we give a description of the experimental setup that was used in order to measure the impact, host memory observations executed from an off-host component can have on the system being observed. The results will be presented in detail in Section 5.4. The full project, containing the source code of our workload generator, as well as all results, is available at `http://sourceforge.net/projects/dmamemoryobserv`.

We used two computers: one being the observer and the other one being the target. Each of them provided a S400 FireWire interface.

Observer: Ubuntu Linux 10.04 64-bit, kernel version 2.6.32, 4GB RAM, Intel Core 2 Quad (Q8200, 2.33GHz).

The establishment of the FireWire connection between the two computers, as well as the access to the main memory and the data transfer, is accomplished by the open source tools published by Boileau [3]. The underlying modules raw1394, IEEE1394, sbp2 (serial bus protocol 2) and OHCI1394 are mandatory and usually distributed as part of the kernel but can also be downloaded from the corresponding repository if not. For the observation of a certain amount of data at specific locations, we wrote a wrapper around Adam

[2] In order to stay within the range, the value stored here needs to be decremented by one.

Boileau's `readWithExclusion()` function which takes the corresponding addresses pointing to the data as an argument.
Target: ARCH Linux 64-bit (a simple and lightweight Linux distribution), kernel version 2.6.33, 2GB RAM, Intel Core 2 Duo (E8400, 3.0GHz).

In our setup, the target's only duty is to run a workload generator and to log the CPU cycles that were consumed while executing it. The workload generator is a C++ console program producing a synthetic workload by continuously writing to a data structure of a pre-defined size. The idea behind this program corresponds to the basic statement of [16]: A forced synchronization between different memory levels (i.e., a cache and the main memory) takes place, whenever two conditions are met: (1) The higher level holds an altered copy $[m]'$ of the original but outdated data $[m]$ which resides in the physical memory. (2) $[m]$ is accessed by another (co)processor. In this case, synchronizations must take place in order to serve the accessing (co)processor with the latest data.

The basic pseudo code of our workload generator is shown in Listing 5.1. The parameters of the `main()` function are:

- *blockSize*: The size (in bytes) of the data structure the workload generator works with.
- *runs*: The number of executions of the `workload()` function.
- *iterations*: The number of times the data structure is written during each execution of the `workload()` function.
- *ratio*: The ratio (i.e., $1 \geq ratio > 0$) expressing the percentage of the data structure that is actually being written.
- *type*: A string, being written to the log file, indicating whether the test ran under observation (o) or normal duty (nd).

Listing 5.1: Pseudo Code of the Workload Generator.

```
1  main(blockSize, runs, iterations, ratio, type)
2  {
3      log.write(printInfo());
4      m = malloc(blockSize);
5      for(i < runs)
6      {
7          startCycles = rdtsc();
8          workload(m, iterations, ratio);
9          endCycles   = rdtsc();
10         log.write(endCycles - startCycles);
11     }
12 }
13
14 void workload(m, iterations, ratio)
15 {
16     for(i < iterations)
17         for(j < m.size()*ratio)
18             m[j] = m[j+1] + 1;
19 }
```

As mentioned in previous work, we propose the observation of relations among critical components of the operating system and its security components. Therefore, we deal with rather small data structures compared to intrusion detection systems which use a signature-based approach. One example here is the observation of entries in the system-call-table, which serves as an interface between user and kernel mode. By altering the function pointers of certain kernel functions or by falsifying their return values, an adversary can successfully conceal the presence of malicious software. As the function names, as well as the function numbers and pointers are rather small, we have decided to run our experiments with a data structure between 8 and 64 bytes.

The number of computations (i.e., the execution of line 18 in Listing 5.1) can be adjusted by the parameters *runs*, *iterations* and *blockSize*. By incrementing the number of *runs* by the

same value the number of *iterations* gets decremented (i.e., `runs += y; iterations -= y`), the number of computations would stay same. But it is important to understand the two major side effects this would have: Experiments showed that $(5 \cdot 10^9) \cdot blockSize$ results in a reasonable runtime for the workload generator when the size of the data structure lies between 8 and 64 bytes. That is, the runtime is high enough to obtain meaningful results even when working with small data structures (i.e., 8 bytes) and low enough to allow an execution of all tests withing one day when working with bigger ones (i.e., 64 bytes). By setting parameter *runs* to $5 \cdot 10^9$ and thus, omitting the first `for` loop in the `workload()` function, we would produce a log file containing $5 \cdot 10^9$ entries; too much data to work with. The second effect is, that by omitting this `for` loop, the time between the two cycle measurements would be very short and, therefore, less meaningful.

The parameter *ratio* relates to the equation for calculating the interference ratio for a given amount of data presented in previous work (Equation 5.1 in this paper). According to [16], the performance loss due to observation depends not only on the amount of data being observed but also on its composition. That is, read-only data cannot be written, therefore, does not need to be synchronized and thus, will not cause resource contention.

$$I = \begin{cases} min, & \text{for } r = 1 \ \vee \ x = 0, \\ max, & \text{for } x = 1, \\ undef, & \text{for } x = 1 \ \wedge \ w = 0, \\ \frac{w \cdot x}{1 - r} \cdot [t], & \text{for } r < 1 \ \wedge \ 0 < x < 1 \end{cases} \tag{5.1}$$

In Equation 5.1, r stands for the ratio of read-only and w for the ratio of writable data ($r + w = 1$). x expresses how much of the processing timespan $[t]$ is actually used for writing data.

The special cases lead to a $minimal$ ($I = 0$) and $maximal$ ($I = 1$) interference rate respectively, while the case of spending all time ($x = 1$) on writing data when there is no writable data available ($w = 0$) is $undefined$.

It is clear to see that the interference ratio I becomes smaller by decreasing the amount of writable data (w) being written (x). And in our setup, this can be done by defining a stress ratio between 0 and 1 which is expressed by the parameter *ratio*, determining the percentage of the workload that is actually being written.

Before every individual test, we log its characteristic, e.g., whether or not the target process ran under normal duty (nd) or was under observation (o). This information is passed by the parameter *type*. In order to get reliable results regarding the execution time of each run, we made use of Intel's Read Time-Stamp Counter instruction `rdtsc` [9]. This assembler command is available since the first Pentium model range and returns the value of a 64-bit model specific register that is incremented every CPU cycle [8]. Furthermore, we disabled the multi-core support and the Intel SpeedStep feature to make sure that our program is carried out by one core only and to keep the maximum CPU clock rate static.

The full experiment included three different cases, characterized by different parameters. Each case C_{1-3} comprised four test tuples T_{1-4} and each test tuple consisted of two tests t_{1-2}. The parameters for each case are given by the signature of the `main()` function in Listing 5.1: While the parameters *runs* (10,000) and *iterations* (0.5mio) stayed the same for all tests making the results comparable, *blockSize* (8, 16, 32, 64), *ratio* (0.5, 1.0) and *type* (o, nd) were adjusted in compliance to the following conditions:
For all cases C_{1-3}:

- The test tuples within the same case are executed with a different *blockSize*: $T_1 = 8$, $T_2 = 16$, $T_3 = 32$ and $T_4 = 64$
- The tests within the same test tuple are executed with a different *type* but inherit the *blockSize* from the corresponding test tuple. Therefore, they are executed with the same *blockSize* but with a different *type*.

- Only one instance of the workload generator executing the tests is running on the target, except for C_2: here, two instances of the workload generator are started, executing the exact same tests in parallel.

- The stress ratio for all tests is 1.0 except for the tests within C_3: here, it is set to 0.5.

The lower bound of the parameter *blockSize* was set to 8, in order to assure a reasonable runtime in conjunction with the agreed on values for the parameters *runs* and *iterations*. All subsequent values are multiples of 8, allowing for an easy comparison of the overall results.

As can be seen, all three cases consist of four experiments where each experiment is conducted with and without concurrent observation. With respect to the conditions above, in the first case, we had only one instance of the workload generator running on the target. For the second case, we started two instances in parallel and observed both at the same time. In either one of them, all bytes allocated were actually written (*ratio* = 1.0). In case three, we again used one instance only but the stress ratio was set to 0.5. In this case only 50% of the allocated data were altered. Once a case was initialized (i.e., all tests were configured and ready for execution), we dumped the full main memory of the target and searched through it, looking for the addresses pointing to our workload. According to the *blockSize* assigned, we then started the observation of the corresponding amount of data whenever parameter *type* = *o*.

5.4 Results

Based on the case conditions introduced in Section 5.3, we obtained meaningful results which allow a direct performance comparison with regard to the CPU cycle consumption of a given process with and without observation. By continuously observing only the actual synthetic workload (i.e., only that amount of data – defined by the parameter *blockSize*– that is actual being worked with) we forced an immediate synchronizations between the caches and the main memory and, therefore, caused a resource contention.

Our results can be seen as a benchmark. As with any other benchmark, it is of prime importance that the results being compared with each other originate from identical experimental setups. Furthermore, the development of CPUs with intrinsic power saving features such as lowering the clock rate or turning off (parts of) certain caches attributes to the fact that reproducing the exact results will most likely be impossible, even if the experiments are carried out by an identical hardware setup. That is, in contrast to, for instance, the Intel SpeedStep feature, which can be seen as a "global" property, future developments will be more fine grained (i.e., instead of a per-socket-feature, a per-core or per-ALU feature will be in place), not allowing any adjustment from the outside. As a result of this, the number of outliers will increase causing a higher standard deviation.

Our results show several clear and generalizable trends and distinctive technical features, expressed by six characteristic values for each test:

- **Process runtime:** This is the time, measured in minutes, executing an individual test took.

- **Measurements:** During each observation, we measured the number of measurements per minute.

- **Mean Value:** As mentioned before, each test returned 10,000 measurements of the consumed CPU cycles. In order to make a statement regarding the average consumption, we calculated the mean value.

- **Standard Deviation:** The standard deviation gives an idea about the level of heterogeneity of all 10,000 measurements per test.

- **Outliers:** A high standard deviation can be an indicator for both a very heterogeneous sample space and a reasonable number of outliers. A closer look into the distribution of our measurements revealed the existence of outliers.

- **Performance Degradation:** The most interesting statement is the one regarding the difference between the CPU cycles consumed by the workload generator with and without observation. This performance degradation is expressed in percentages.

Before we present the three generalizable trends according to the characteristic values from above, we want to describe the results of one sample case, namely the one where we observed one process only, with a stress ratio of 1.0. Since the results of all other cases are of a similar character, the in detail presented example shows the effects of observations carried out by a coprocessor very clear.

5.4.1 Sample Results: Observing One Process with a Stress Ratio of 1.0

As mentioned before, one case consisted of eight tests, where each two tests of one tuple are characterized by working on a data structure of the same size but are of a different type. Table 5.1 shows the conditioned results obtained from all tests within case one (one process instance, *ratio* = 1.0). Column one (runtime in minutes) represents the runtime of each test. The figures in the second column (measurements per minute) stand for the frequency with which the observation of the data structure – characterized by the size shown in column three (size) – was executed. The zeros in the second column indicate that this test ran without observation (parameter *type* = nd). Column four (performance degradation) depicts the percentage values of the difference between two tests of the same test tuple, based on the averaged CPU cycles shown in column five (mean value). The number of outliers, stated in column six, will be discussed shortly. They are closely related to the standard deviation shown in column seven.

The high standard deviations shown in the last column of Table 5.1 are due to the fact that our data is distinguished by a number of very clear outliers. Using the results that lead to the values shown in the first row of Table 5.1, we created a histogram that clearly reveals the outliers. Figure 5.1 sharply uncovers that 98.19% of all values lie within the range of 53mio - 54mio CPU cycles. Further investigation showed that the outliers appear almost periodically which strengthens our suspicion that these exceptionally high values are caused by processes belonging to the operating systems running on the target, demanding for processor time. Table 5.2 depicts the normalized results of the first test. That is, we eliminated all outliers in order to show that the actual result set is quite homogeneous. The still notably higher standard deviation of the normalized results for the last test tuple is due to a higher degree of heterogeneity in the corresponding result set, which increases with

Table 5.1: All conditioned results obtained from observing one process with a stress ratio of 1.0.

Runtime in min	Measm. min	Size	Perf. Degr.	Mean Value	Outlier	Std. Div.
3.07	0	8	0.77%	54.99mio	180	36.24%
3.08	130,768			55.41mio	183	36.26%
4.97	0	16	1.28%	88.90mio	294	28.48%
5.02	69,887			90.04mio	298	28.30%
8.15	0	32	1.48%	146.04mio	480	21.94%
8.25	34,076			148.20mio	491	21.84%
15.57	0	64	16.39%	279.41mio	928	15.63%
18.12	16,881			325.19mio	1,085	14.34%

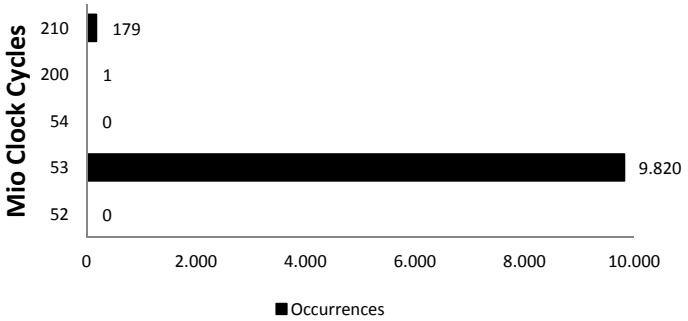


Figure 5.1: Histogram of all results from the first test (corresponding to row one in Table 5.1).

Table 5.2: Normalized conditioned results for observing one process with a stress ratio of 1.0.

Performance Degradation	Mean Value	Standard Deviation
0.72%	52.29mio	0.04%
	52.67mio	0.10%
1.27%	84.50mio	0.02%
	85.57mio	0.02%
1.44%	138.85mio	0.02%
	140.85mio	0.18%
16.36%	265.45mio	1.47%
	308.94mio	1.26%

the amount of data being processed by the observer. According to the characteristic values in Table 5.1 and 5.2, we derived three generalizable trends described in the following subsections.

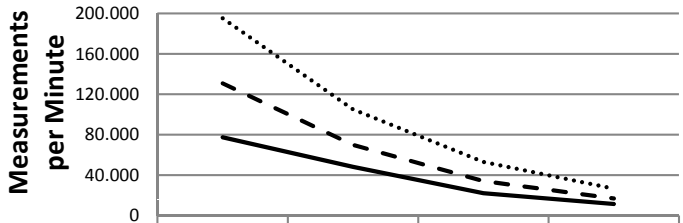
5.4.2 Observing Frequency vs. Size of the Data Structure

As mentioned earlier, the size of the data structure we observed was set to either 8, 16, 32 or 64 bytes by the parameter *blockSize*. While the amount of data being subject to observation rises, the frequency with which the data can be observed decreases *almost* proportionally: While we were able to observe 8 bytes with an average frequency of 195,201 measurements per minute, twice the amount of data (16 bytes) could only be observed with roughly half of the frequency (105,126 measurements per minute). This behavior is depicted in Figure 5.2. The explanation for this effect is rather straightforward: Observing data from a coprocessor implies the task of copying the concerned data from the host towards a memory closer to the coprocessor. Transferring small chunks of data is just faster than transferring bigger ones. And as we have only one DMA channel between our target and the observer, a new observation can only start after the previous one has finished.

For interpreting Figure 5.2 correctly, it is important to know that the lines connecting the measuring points have just been added in order to emphasize the trend and to facilitate the readability. The impression that the frequency degradation is almost linear is deceitful.

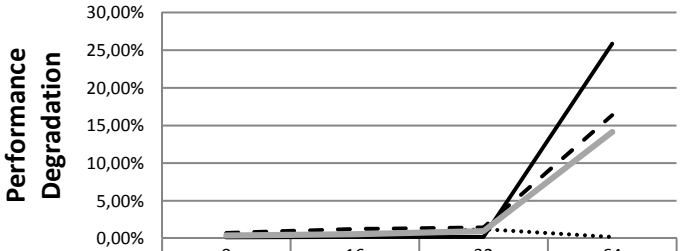
5.4.3 Performance Degradation

Even though the details of Figure 5.3 may mislead to a slightly different conclusion, the averages calculated from all values (depicted by the solid gray line) clearly show: The larger the data structure we observe, the bigger the resource contention on the host being



	8	16	32	64
1P 50%	195.201	105.126	53.017	26.641
1P	130.768	69.887	34.076	16.881
2P	77.385	48.173	22.161	11.394

Figure 5.2: Measurement frequency for different data structure sizes.

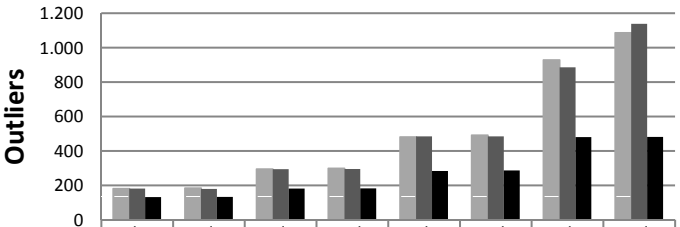


	8	16	32	64
1P 50%	0,30%	0,23%	1,25%	0,21%
1P	0,75%	1,27%	1,46%	16,37%
2P	0,16%	0,22%	0,20%	25,84%
Avrg.	0,40%	0,57%	0,97%	14,14%

Figure 5.3: Performance degradation according to the size of the observed data structure.

subject to observation. The percentaged values shown in Figure 5.3 are the averages calculated from the performance degradation with and without outliers. The message this figure sends out is clear: Observations have a significant performance impact on the host. While the measurements for the case of observing one or two processes with a stress ratio of 1.0 (i.e., the workload generator works on 100% of the initialized data) confirm this message, the curve describing the case of observing one process with a stress ratio of 0.5 seems to contradict it. That is, while executing the tests with a data structure of 64 bytes and a stress ratio of 0.5, the performance impact decreased drastically. This effect is most likely the result of system-based algorithms dealing with the assembling of cache lines, working sets, etc. by using techniques such as pre-fetching. Since our way of measuring the performance impact is an end-to-end approach, the explanation of such effects is out of scope. We leave aside the exact diagnosis of intrinsic impacts caused by protocol overheads or diverse bridges and controllers our signal passes, as well as the performance boost a process can experience when the executing processor benefits from a good locality of reference within the data it demands. Therefore, we measure what is important in practice: the degree of performance degradation including all side effects.

The downside of this approach is that we can only speculate regarding the causes of certain effects. Special equipment as used by chip vendors would have been necessary in order to further investigate on these effects. The important remark here: The results stay the same, even if one would be in the position of giving a detailed description on the internal behavior.



	8/nd	8/o	16/nd	16/o	32/nd	32/o	64/nd	64/o
1P	180	183	294	298	480	491	928	1.085
2P	182	180	295	295	485	485	886	1139
1P 50%	132	133	181	182	284	287	481	482

Figure 5.4: Increasing number of outliers.

5.4.4 Increasing Number of Outliers

In Section 5.4.1 we already presented a closer look into the outlier effect and used the results of our first test (row one in Table 5.1) to show that besides the easily identifiable outliers, our result set is heterogeneous. Figure 5.4 now depicts the growth of the number of outliers with regard to the size of the data structure the workload generator works with. Three major characteristics are important to point out: The first one is that our observation causes the number of outliers to increase. While for the cases where we observed smaller data structures the difference was marginal, the gap rose when we increased them. Interesting to know here is the fact that the number of outliers has only a very small influence on the performance degradation[3] as one can see by comparing the corresponding values shown in Tables 5.1 and 5.2. The second important point regards to the growth of the absolute appearances of outliers. That is, the more data we work with, the more outliers appear. And last but not least, one can see that working with only half the data initialized caused roughly half the amount of outliers. Even though the outliers are clearly to identify in all cases and thus can be eliminated for statistical purposes, they are an intrinsic element of the workload process running on the target. The avoidance of such effects may be desirable but due to several reasons (e.g., multi-tasking on one processor) not always possible.

5.5 Conclusion

We have shown that memory observation as may arise in concurrent observation for intrusion detection and prevention, implies a non-negligible cost – particularly when executed over a shared memory or non-uniform memory architecture. Validating a model for such NUMA architectures, this paper has shown that data structure size and the fraction of write activities is clearly having effects that cannot be fully masked by even advanced processor and memory architectures.

Memory observations executed by a coprocessor imply *costs* (i.e., performance degradation) on the side of the host being observed. And in this work we were able to quantify these costs for a given hardware. [16] explain that the reasons for this loss of performance can be found in the memory architecture of today's computers: Data most recently used by a processor is copied to a memory level which is much smaller than the systems main memory but also much faster. The probability that data used at time t_0 is needed at time t_1 makes this approach very efficient. Whenever the copied data gets altered, synchronization between the different memory levels involved (i.e., in most architectures L1-cache, L2-cache and main memory) does not take place immediately. Instead a coherence protocol is implemented, triggering the synchronization according to a set of rules. One of these

[3] Because of this, we were able to average the values for the performance degradation shown in Figure 5.3.

rules is that synchronization must take place when another processor is about to access data residing in the main memory that is marked as *dirty*. That means that there exists an altered copy of this data in a higher memory level. In order to fulfill the requirement of serving every processor with the most current data, synchronization is indispensable. And since the accessing speed of the different memory levels differ potentially by one or more orders of magnitude, each synchronization results in a loss of performance. A cost model for expressing the worst case with regard to the performance degradation such synchronizations can cause has been presented in previous work. Its practical and quantitative validation was missing until now.

The results obtained from our experiments fully validate the theoretical statements proposed previously. We implemented a workload generator and executed it on a lightweight Linux operation system (i.e., ARCH Linux) in order to see the effects as clearly as possible. The actual observation was realized using a high-speed bus-mastering DMA channel established between a second computer and the target system. This corresponds to the prerequisite of [16] to have the observation executed by another processor than the one operating the host system (i.e., a coprocessor). By running tests for three different cases (eight tests per case) where each test was distinguished by the parameters with which it has been executed, we clearly unveiled the performance impact. With the cases executed in our test environment (cf. Section 5.3), we experienced a performance degradation of more than 25% for the case where we observed two instances of our workload generator running in parallel and working on a data structure of 64 bytes each.

The size of the data structure being observed is not the prime factor leading to a well-founded assumption of whether or not the loss of performance is expected to be rather high or low. That is, the composition of the data structure plays a vital role: As already mentioned, synchronizations take place, when altered cached data needs to be written back to the main memory. If the cached data is either read-only or writable but not being written, its value will never change and thus, synchronizations are not needed. Together with this statement, an equation capable of calculating the inference rate was published (cf. Equation 5.1), basically saying that the observation of read-only data is *cheap* in comparison to the observation of writable data, that is actually being written. By setting the stress ratio for one case to 0.5, we achieved that the workload generator worked on only 50% of the data initialized.

It can clearly be seen that the results validate the equation presented in previous work, as – compared to the cases with a stress ratio of 1.0 – the performance impact on the target was significantly lower as shown in Table 5.3. Here, we juxtaposed the percentaged performance degradations for the case of observing one instance of the workload generator working on 50% and 100% of its workload. Just as in Figure 5.3 the percentages shown in

Table 5.3: Comparison of performance degradations depending on the amount of data actually being written.

Size	**Performance Degradation**	
	ratio = 0.5	***ratio*** = 1.0
8	0.30%	0.75%
16	0.23%	1.27%
32	1.25%	1.46%
64	0.21%	16.37%

Table 5.3 are the averages obtained from calculating the performance degradation with and without inclusion of outliers. It is clear to see that the performance degradation is much lower when not 100% of the data being observed is altered (tested with 50% and 100% in our case).

We have shown that memory observation executed by a coprocessor does not come for free. In fact, depending on the size and composition of the data structures being subject

to observation, the performance degradation varies: Bigger data structures and a higher appearance of data being written result in a more serious loss of performance.

With regard to future work, the results represented in this paper are essential. They prove that an observation strategy according to the busy-wait approach would result in a tremendous loss of performance and is, therefore, not desirable. This is especially true when we consider that the results presented in this work were obtained using the comparatively slow S400 FireWire technology (400Mbit/s), whose interface is bound to the input/output controller hub (southbridge) while the host's main memory is connected to the memory controller hub (northbridge). This resulted in signal latency and caused the maximum measurement frequency to be rather low. The maximum measurement frequency that could have been achieved by using a highly parallel and multi-threaded multi-core GPU – connected to the northbridge – would have been much higher. Thus, causing an even bigger performance degradation when following the busy-wait approach is likely.

5.6 Future Work

The results reported in the present work were obtained from a single observer and target tuple. In order to rule out anomalies arising from this particular configuration, ongoing experiments are being conducted with further platforms.

In our experimental setup we took advantage of the FireWire S400 technology which served as an easy to use approach to observe the data structures on the target. This was done with a maximum of 195,201 measurements per minute. Besides the bandwidth limitation as such (400Mbit/s), the fact that our signal had to pass two south- and northbridges (target and observer) accounts for the relatively low observation frequency. In our future work, we will overcome the limitations revealed by [14] and use the GPU (up to 16GB/s) for our purposes. By doing so, the measurement frequency will benefit from a higher bandwidth due to the fact that this coprocessor is directly connected to the same integrated memory controller the main memory is coupled with.

By designing smart algorithms which are capable of exploiting the parallel design of modern graphics cards, we will propose an observation model that takes advantage of the maximum available observation frequency if necessary but usually tries to work as efficient as possible. That is, keeping the frequency as low as possible while still guarantying a reasonable probability that a subversion is detected while taking place.

Acknowledgment

The authors would like to thank Julian Knauer, Pierre Schnarz and Sufyan Siddique for their excellent practical support and Reinhard Riedmüller for his comments on the final draft.

Bibliography

[1] APPLE COMPUTER, INC., COMPAQ COMPUTER CORPORATION, INTEL CORPORATION, MICROSOFT CORPORATION, NATIONAL SEMICONDUCTOR CORPORATION, SUN MICROSYSTEMS, INC. AND TEXAS INSTRUMENTS, INC. 1394 – Open Host Controller Interface Specification, Jan. 2000. (Last checked: August 22, 2012). 68

[2] BOILEAU, A. Hit by a Bus: Physical Access Attacks with Firewire, 2006. (Last checked: August 22, 2012). 69

[3] BOILEAU, A. Firewire, DMA & Windows, Mar. 2008. (Last checked: May 15, 2012). 69

[4] CHO, Y. H., AND MANGIONE-SMITH, W. H. A Pattern Matching Coprocessor for Network Security. In *Proceedings of the 42nd Annual Design Automation Conference (DAC*

2005) (Anaheim, CA, USA, June 2005), ACM, pp. 234–239. (Last checked: August 22, 2012). 27, 67

[5] DORNSEIF, M. 0wned by an iPod, Nov. 2004. (Last checked: August 22, 2012). 22, 69

[6] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-1995* (1996). doi:10.1109/IEEESTD.1996.81049. 68

[7] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-2008* (2008), 1–906. doi:10.1109/IEEESTD.2008.4659233. 68

[8] INTEL. Using the RDTSC Instruction for Performance Monitoring. Tech. rep., Intel Corporation, 1997. (Last checked: August 22, 2012). 71

[9] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. Tech. rep., Intel Corporation, Mar. 2010. (Last checked: August 22, 2012). 71

[10] MCEVOY, T. R., AND WOLTHUSEN, S. D. Concurrent Host Integrity Protection and Intrusion Detection. Unpublished early draft, 2009. 44, 45, 54, 57, 67, 91, 93

[11] PASKINS, A. The IEEE 1394 bus. In *Proceedings of New High Capacity Digital Media and Their Applications* (May 1997), pp. 4/1–4/6. (Last checked: August 22, 2012). 9, 68

[12] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[13] QUINN. FireStarter, 2002. (Last checked: August 22, 2012). 69

[14] RIEDMÜLLER, R., SEEGER, M. M., WOLTHUSEN, S. D., BAIER, H., AND BUSCH, C. Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection. In *Proceedings of the 2nd International Workshop on Security and Communication Networks (IWSCN 2010)* (Karlstad, Sweden, May 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/IWSCN.2010.5497999. 6, 8, 9, 10, 12, 68, 78, 93, 94, 96

[15] RUTKOWSKA, J. Beyond The CPU: Defeating Hardware Based RAM Acquisition, Feb. 2007. (Last checked: August 22, 2012). 58, 62, 69, 95

[16] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[17] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium of Recent Advances in Intrusion Detection (RAID 2008)* (Boston, MA, USA, Sept. 2008), vol. 5230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 116–134. doi:10.1007/978-3-540-87403-4_7. 28, 67, 92

[18] WILLIAMS, P. D., AND SPAFFORD, E. H. CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection. *Computer Networks 51*, 5 (Apr. 2007), 1284–1298. doi:10.1016/j.comnet.2006.09.011. 30, 43, 67, 93, 94

[19] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure Coprocessor-Based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, July 2002), ACM, pp. 239–242. doi:10.1145/1133373.1133423. 23, 24, 43, 53, 54, 59, 61, 62, 67

Chapter 6

Using Control-Flow Techniques in a Security Context A Survey on Common Prototypes and their Common Weakness

Abstract

Practical approaches using control-flow techniques in order to detect changes in the control-flow of a program have been subject of many scientific works.

This work focuses on three common tools making use of control- and data-flow analysis in order to detect alternations and reveals their common weakness in terms of the ability to react directly to a dynamic change in control-flow. With a general focus on static analysis of binaries or source code, dynamic changes in the executive flow cannot be detected. In order to emphasize this shortcoming of static analysis, we present an approach for dynamically changing a program's control-flow and validate it by depicting a proof of concept.

6.1 Introduction

Analyzing the flow of control and data within a program is a well-known technique intrinsic to the field of compiler construction. While program optimization is one of its main fields of application, the field of security with regard to information technology (IT) has chronicled the appearance of control- and data-flow analysis years ago. The appearance of subversion techniques concealing their malicious intents by altering the program flow contributed its part to this development. Nowadays, buffer overflow attacks are the widest spread representative of exploiting the deviation of control-flow. Here, software vulnerabilities are used in order to inject and execute defective code. Thus, detecting such attacks is possible and practical approaches able of countering them have been presented by various authors.

We took a closer look at three common prototypes which make use of flow analyzing techniques in order to detect a change in the executive flow of a program. The result of the investigation is subject of this work and reveals that all of them follow a static concept. That is, they are unable to react to dynamic changes in control-flow.

To this extent, we developed a C++ program making use of a Linux system call mostly utilized to implement breakpoint debugging in order to take advantage of a characteristic of the late binding technique, which is commonly used for loading dynamic link libraries (DLL). In a testing environment, this proof of concept was able to change the control-flow of a target program while staying undetected by major anti-virus suits.

The remainder of this work is structured as follows: Section 6.2 presents a short history on control- and data-flow techniques in general before Section 6.3 goes into detail by outlining the major features of control-flow (6.3.1) and data-flow (6.3.2) analysis. The brief presentation of the three selected common prototypes is subject of Section 6.4. Section 6.5 depicts our prototype by describing the approach as such, as well as showing some pseudo

code. The paper is closed with a conclusion in Section 6.6 and an outlook into the future work in Section 6.7.

6.2 Related Work

Control-flow analysis is not new at all. Under the name of "boolean matrices" it was already mentioned in papers published in the second third of the 20^{th} century with the earliest one being from Lefschetz, published in 1930 [10]. The idea behind this special kind of matrices evolved from the need to have the data of flow diagrams available in such a way that it could be handled by machines. Likewise to today's ambitions, this data was used for program analysis with regard to internal consistency and the identification of subroutines [15].

Extensive research has been done in this field with regard to the applicability of control-flow analysis in a security related context. In general, the control-flow of a given program is studied in order to tell apart normal from abnormal behavior; to say this with the words of Forrest et al.: "The problem of protecting computer systems can be viewed generally as the problem of learning to distinguish *self* from *other*." [5]. With respect to this, Wespi et al. (cf. [19]) proposed a technique for creating patterns that can be used to model normal behavior of a given process. These models represent the *self* and can be used for intrusion detection under the assumption that the *other* can be identified by a behavior different from what was used to train the model.

Enforcing control-flow for security reasons is – amongst others – the subject of [14] and [17]. While the former present a binary rewriting approach in order to augment existing programs, the latter present an architectural mechanism in which the processor tracks the information flows in question and, therefore, is capable of detecting dangerous uses of spurious values.

Abadi et al. claim that the enforcement of control-flow integrity (CFI) "...cannot be subverted or circumvented even though it applies to the inner workings." [1]. In their paper, they show the practical prove that CFI enforcement is capable of ensuring that runtime execution of a given program proceeds along a given control-flow graph[1] (CFG).

While for intrusion detection purposes both techniques are used, control-flow and data-flow analysis, [11] shows how the gap between them can be bridged. They present practical results on how data-flow analysis can be used in order to leverage the results from control-flow analysis.

In [13] the authors demonstrate that static analysis for intrusion detection is no longer a sufficient technique for identifying malware. They use binary code obfuscation in order to circumvent detection by semantics-based malware scanners.

6.3 Flow Analysis

Flow analysis comes in all kind of flavors – with and without relevance to information security. In this section, we focus on the two different techniques of control-flow and data-flow analysis. While both techniques were initially used for optimization reasons (and still are), they are nowadays also used in the field of intrusion and malware detection.

As we will see, the two presented analyzing techniques are not mutually exclusive as one can be used as a preprocessor for the other.

6.3.1 Control-Flow Analysis

From a very abstract point of view, a program is nothing else than a series of instructions that can be addressed. The decision as to which instruction is to be executed next depends

[1] A software called "Vulcan" [16] was used to gain the control-flow graphs.

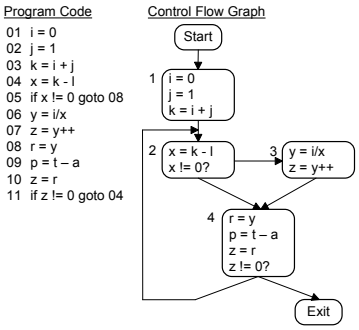


Figure 6.1: Example of a control-flow graph.

on the result of what is currently happening. By analyzing the source code of a program or its binary, all instructions can be identified and a CFG containing all possible flows of control can be retrieved. A CFG contains nodes and edges. Each node represents what is called a basic block: a maximum set of instructions without any possibility to merge or fork.

There is an edge between two nodes if the last instruction of a basic block leads to the start of another basic block. A simple version of a CFG is shown in Figure 6.1. As in a CFG there is usually more than one possible path leading from the start of the program to its final exit, an automated way of analyzing this structure is needed. This is the subject of control-flow analysis.

6.3.2 Data-Flow Analysis

As the name already suggests, data-flow analysis is rather concerned with the data that flows within a program than its control. With respect to information security, the meaningfulness of CFGs has been proven in theory and practice and its analysis is known to be effective. Data-flow analysis on the other hand is a discipline whose history started in 1977 when Hecht published his book on *Flow Analysis of Computer Programs* [8].

Data-flow relates to the arguments of system calls and to the actual data (i.e., variables) that a specific program deals with. With respect to information security, monitoring the system call arguments in contrast to analyzing CFGs of a given program, comes with the advantage that malicious behavior can be detected even in cases where the actual control-flow stays unmodified.

Generally data-flow analysis is concerned with the question of the lifetime of variables. As the states of all variables may never be known and since the input of one basic block is the output of its predecessor, data-flow analysis is done by approximating the values of variables coming into a specific basic block. The calculation is then done by feeding these values into iterative algorithms such as the round robin algorithm in order to find so-called fix points. That is, values for the incoming variables that satisfy the function $f : X \to X$.

Using data-flow analysis for anomaly detection is the subject of [2]. In their work, the authors present an algorithm that can be layered on top of existing programs that rely on control-flow techniques for being invulnerable against attacks such as non-control-flow hijacking attacks where, for instance, configuration data or user input is used in order to exploit vulnerabilities of certain applications.

Figure 6.2 shows a sample data-flow graph (DFG). For a better comparability between CFG and DFG, the DFG is based on the same synthetic program code as the CFG shown in Figure 6.1.

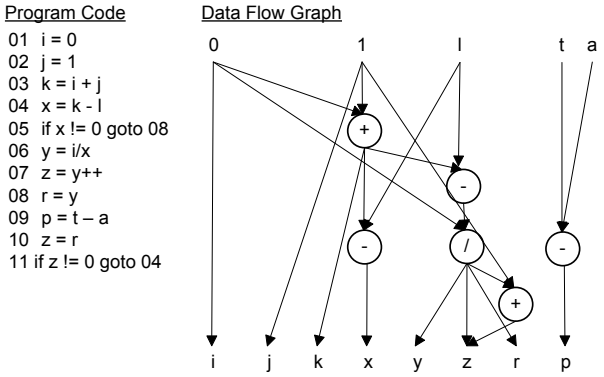


Figure 6.2: Example of a data-flow graph.

6.4 Common Prototypes Using Flow Analyzing Techniques in a Security Context

In this section, we present three common software prototypes which use flow analyzing techniques for security purposes.

6.4.1 Binary Executable Structurizer

In [18] a "lightweight assembler structural representation tool" called BEST (Binary Executable Structurizer) is introduced. It incorporates the use of control- and data-flow analyzing techniques to analyze the executive flow of a program. As the name already suggests, BEST does not rely on the source code of the programs in question as it works with the program binaries. Besides the recovery of the control- and data-flow, the tool also represents the assembler of a binary in a "vulnerability-mining-oriented intermediate language" called PANDA. In contrast to plain assembler, a PANDA representation of assembler code is easier to assess and thus, facilitates further security analysis. Since PANDA's basic control structures, as well as mathematical and logical operations are taken from the C programming language, a program represented in PANDA looks much like a C program with some embedded assembler syntax.

In order to retrieve the flow graphs from a binary, BEST relies on third party programs, namely IDA Pro (cf. [9]) and Objdump (cf. [6]). Based on the output of these tools (either one or the other is used), the BEST algorithms start working. Besides the PANDA code, BEST also outputs a control-flow graph and a control tree graph (shows the nesting relationship of control structures). With the PANA code available, security professionals can then assess the binary in question in order to unveil security flaws.

6.4.2 Control-Flow Security Analysis Approach

In [4] a control-flow security model (CFSM) which allows program execution to dynamically follow the statically declared security properties specified as a control-flow constraint specification (CFCS) is introduced. This approach counters, for instance, buffer overflow attacks, which have the intention of diverting program execution by jumping to previously injected malicious code. Thus, illegally altering the execution flow of programs in order to control the software behavior. The proposed model includes the formal definition for program semantics and security properties for control-flow. With the CFSM applied to source code or binaries, execution paths violating the predefined CFCSs can be identified and prevented. This is done by ensuring two security properties which regard to data memory

Table 6.1: Characteristics of the presented flow analyzing tools.

Tool Name	Uses CFG	Uses DFG	Operates Statically	Reads Binary	3^{rd} Party Tools
BEST	✓	✓	✓	✓	✓
CDSM	✓	✗	✓	✓	✗
SAFE	✓	✗	✓	✓	✓

that is not executable and code memory that is not writable. In the latter case, observations due to loading dynamic link libraries are excluded.

The control-flow graph needed to define the CFSM is retrieved applying a two step approach starting with a conservative CGF based on the analysis of function calls and the branch jumps of basic blocks, followed by a static analysis to refine and improve the former outcome. The overall result is a graphical control-flow security model annotated with the corresponding control-flow constraints.

6.4.3 Static Analyzer for Executables

In addition to an architecture that is resilient to common obfuscation transformations and capable of detecting malicious patterns in x86 executable binaries, [3] also introduce the proof of their concept by implementing SAFE, a static analyzer for executables. The heart of SAFE – namely the detection component – takes two arguments. One is an annotated control-flow graph of the executable and the other is a generalization of malicious code (i.e., a representation of obfuscated strains of a virus). The CFG itself is the result of the annotation module, which uses preloaded pattern definitions to annotate the plain CFG gained from a combinational use of the third party tools IDA Pro (cf. [9]) and CodeSurfer (cf. [7]). While Norton AntiVirus, McAfee VirusScan, and Command AntiVirus were not able to detect any of the four obfuscated[2] viruses during the practical experiments, SAFE was able to identify 100% of them and thus, has a false rate of zero.

6.4.4 Review of the Presented Prototypes

In the preceding subsections, we gave a survey of three common prototypes which are using flow analyzing techniques in a security related context. While BEST and the CFSM approach are from 2010 and 2009, respectively, SAFE was presented in 2003. Table 6.1 gives an overview over five main characteristics of the tools presented in Section 6.4. All of them use control-flow techniques while only BEST combines control- and data-flow. Also, all tools perform a static analysis on binary executables. Only the CFSM approach waives 3^{rd} party tools while both BEST and SAFE depend on the quality of the outcome of tools such as IDA Pro (cf. [9]), CoderSurfer (cf. [7]) or Objdump (cf. [6]).

BEST is not a fully automated tool. Its result is a "vulnerability-mining-oriented intermediate language" [18] of the binary in question, which serves the purpose of easing the process of program code understanding. Thus, reacting automatically to ongoing security breaches is not possible.

The CFSM approach is able to detect any exploit that alters the execution paths of the binary in question. While the approach presented is a reasonable method in order to detect buffer overflow attacks, it comes with one major restriction: Alterations made directly or indirectly due to loading dynamic link libraries cannot be detected. SAFE, which is able to detect obfuscated viruses with a much higher rate of reliability than Norton AntiVirus, McAfee VirusScan, and Command AntiVirus relies on both the quality of the output of 3^{rd} party tools and the completeness of the pattern definitions. All three approaches have one drawback in common: They are not designed to instantly react to changes in the executional flow caused by dynamically loaded code.

[2]Obfuscation technique: code transposition

6.5 Example of Dynamically Altering Control-Flow

In this section, we will present our approach which is capable of dynamically altering the execution flow of arbitrary executables[3] using late binding techniques for loading dynamic link libraries. The according proof of concept is implemented as three C++ files. Listing 6.1 depicts the target program which is the subject of our attack[4].

Listing 6.1: Pseudo Code of the Target Program.

```
1 main()
2 {
3     int  ret              = 0;
4     char *dllName         = "myDll";
5     void *handle          = dlopen(dllName);
6     int (*isValidUser)()  = dlsym(handle, "isValidUser");
7
8     ret = isValidUser();
9 }
```

Listing 6.2: Pseudo Code of the DLL.

```
1  extern int isValidUser()
2  {
3      if (userId >= 1000) {
4          return 1;
5      }
6      doAdminStuff();
7      return 0;
8  }
9
10 void doAdminStuff()
11 {
12     ...
13 }
```

In line 6 (Lst. 6.1), we create a function pointer pointing to a function named `isValidUser()`, implemented in a dynamic link library. This function is called in line 8. In line 3 of our DLL, shown in Listing 6.2, the `userID` of the current user is checked. Only if it is lower than 1,000 a function named `doAdminStuff()` is executed. The original control-flow is depicted as solid-lined arrows in Figure 6.3.

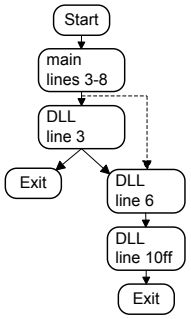


Figure 6.3: CFG of the target program (Lst. 6.1) and its DLL (Lst. 6.2).

[3] While our proof of concept presented here regards to the Linux operating system, we also have an equivalent version running on Windows.

[4] In the field of offline computer games, software that implements such features in order to alter game specific values (e.g., health, energy, ammunition, etc.) is called a trainer.

Listing 6.3: Pseudo Code the Program Executing the Exploit.

```
1 main(int argc, char *argc[])
2 {
3     int pid         = argc[0];
4     long oldPointer = argc[1];
5     long newPointer = argc[2];
6
7     ptrace(PTRACE_ATTACH, pid, NULL, NULL);
8     ptrace(PTRACE_POKEDATA, pid, oldPointer, newPointer);
9     ptrace(PTRACE_DETACH, pid, NULL, NULL);
10 }
```

The exploit code is shown in Listing 6.3. It is started with three parameters, namely the process id, the address of the function to be attacked (`oldPointer`) and the address of the new function (`newPointer`). `oldPointer` is the address the function pointer `(*isValidUser)()` which points to the address of the variable on the heap and `newPointer` is the address of the new function (`doAdminStuff()`). In order to achieve our goal of altering the control-flow dynamically, we make use of the `ptrace()` system call, that "...provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers." [12]. This method is mainly used for implementing breakpoint debugging and system call tracing. In line 7 we start tracing the target process. As a result of this, we are from now on the parent of the target. While the output of the `ps()` command reflects this change in ownership, issuing the `getppid()` command would reveal the original parent. Before we are undoing the reparenting effect in line 9, we perform the actual attack in line 8: Here, we overwrite the address of the old function with the address of the function we want to execute instead. With this being done, we dynamically changed the control-flow. The function call in line 8 of Listing 6.1 does not point to line 1 of Listing 6.2 anymore, thus, the check of the user id in line 3 is circumvented. Instead function `doAdminStuff()` is executed directly. The altered control-flow is depicted as a dashed-lined arrow in Figure 6.3, completely bypassing basic block 2. This attack works and stays undetected because it takes advantage of a specific characteristic of the late binding technique. When late binding is used, the affected variables are stored on the heap. They stay writable since they receive their actual value during runtime. Our approach exploits this fact and uses a system call intended for debugging purposes in order to overwrite the value of a specific variable stored on the heap.

Being able to detect such attacks is a matter of distinguishing "*self* from *other*" [5].

6.6 Conclusion

We reviewed three common practical tools which make use of control-flow graphs in a security related context. Specific to all of them is the fact that they operate statically and thus, are unable to react to dynamically undertaken changes of control-flow. To this extend, we have presented a proof of concept taking advantage of a Linux system call, mainly used for debugging purposes, and a characteristic of the late binding technique, where the corresponding variables are placed on the heap. We showed pseudo source code of our exploit and pointed out that altering the control-flow of a program which makes use of the late binding technique (e.g., loading of DLLs) is possible. Countering such an attack is not an easy task since it does not use any vulnerability. Therefore, counter measurements are fronted with the problem of telling apart legal alterations of the value of variables stored in the heap and illegal ones – the subject of anomaly detection.

6.7 Future Work

Our future work will be twofold. We will be concerned with developing a proof of concept which will perform observation tasks on variables on the heap. In compliance with a pre-defined security policy, any changes on values ultimately resulting in a change of executive flow will be brought to the attention of the user who will be given the ability to allow or

block the action. In addition to that, we will concentrate on a novel concept allowing for automatically made decisions regarding the degree to which an alteration of control-flow regards to *self* oder *other* (i.e., is legal of illegal).

Acknowledgment

The author would like to thank Julian Knauer for his excellent practical support.

Bibliography

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC) 13* (Nov. 2009), 4:1–4:40. doi:10.1145/1609956.1609960. 82

[2] BHATKAR, S., CHATURVEDI, A., AND SEKAR, R. Dataflow Anomaly Detection. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2006)* (Oakland, CA, USA, May 2006), IEEE Computer Society, pp. 48–62. doi:10.1109/SP.2006.12. 83

[3] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (SSYM 2003)* (Washington, DC, USA, Aug. 2003), USENIX Association, pp. 169–186. (Last checked: August 22, 2012). 85

[4] CHUNLEI, W., GANG, Z., AND YIQI, D. An Efficient Control-Flow Security Analysis Approach for Binary Executables. In *Proceedings of the IEEE International Conference on Computer Science and Information Technology (ICCSIT 2009)* (Beijing China, Aug. 2009), IEEE Computer Society, pp. 272–276. doi:10.1109/ICCSIT.2009.5234950. 84

[5] FORREST, S., PERELSON, A. S., ALLEN, L., AND CHERUKURI, R. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1st IEEE Symposium on Security and Privacy (S&P 1994)* (Oakland, CA , USA, May 1994), IEEE Computer Society, pp. 202–212. doi:10.1109/RISP.1994.296580. 10, 82, 87

[6] GNU BINUTILS. ObjDump. (Last checked: August 22, 2012). 84, 85

[7] GRAMMATECH. CodeSurfer. (Last checked: August 22, 2012). 85

[8] HECHT, M. S. *Flow Analysis of Computer Programs*. Elsevier Science, 1977. 83

[9] HEX-RAYS. IDA Pro. (Last checked: August 22, 2012). 84, 85

[10] LEFSCHETZ, S. Topology. *American Mathematical Society XII* (1930). 82

[11] LI, P., PARK, H., GAO, D., AND FU, J. Bridging the Gap between Data-Flow and Control-Flow Analysis for Anomaly Detection. In *Proceedings of the 24rd Annual Computer Security Applications Conference (ACSAC 2008)* (Anaheim, CA, USA, Dec. 2008), IEEE Computer Society, pp. 392–401. doi:10.1109/ACSAC.2008.17. 82

[12] LINUX MANPAGE. ptrace(). (Last checked: August 22, 2012). 87

[13] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Computer Society, pp. 421–430. doi:10.1109/ACSAC.2007.21. 10, 43, 53, 82

[14] PRASAD, M., AND CHIUEH, T. A Binary Rewriting Defense against Stack Based Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference* (San Antonio, TX, USA, June 2003), USENIX Association, pp. 211–224. (Last checked: August 22, 2012). 82

[15] PROSSER, R. T. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Proceedings of the Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, MA, USA, Dec. 1959), ACM, pp. 133–138. doi:10.1145/1460299.1460314. 82

[16] SRIVASTAVA, A., EDWARDS, A., AND VO, H. Vulcan – Binary transformation in a Distributed Environment. Tech. rep., Microsoft Research, 2001. (Last checked: August 22, 2012). 82

[17] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)* (Boston, MA, USA, Oct. 2004), ACM, pp. 85–96. doi:10.1145/1024393.1024404. 82

[18] WANG, W. BEST: An Assembler Structural Representation Tool Based on Flow Analysis. In *Proceedings of the International Conference on Management and Service Science (MASS 2010)* (Wuhan, China, Aug. 2010), IEEE Computer Society, pp. 1–4. doi:10.1109/ICMSS.2010.5575669. 84, 85

[19] WESPI, A., DEBAR, H., DACIER, M., AND NASSEHI, M. Fixed- vs. Variable-length Patterns for Detecting Suspicious Process Behavior. *Journal of Computer Security 8*, 2,3 (Aug. 2000), 159–181. (Last checked: August 22, 2012). 82

Chapter 7

Towards Concurrent Data Sampling using GPU Coprocessing

Abstract

Host intrusion detection systems operating on the host under observation itself are limited by an adversary's ability to subvert all data collection and the detection and mitigation mechanisms themselves. Although coprocessor architectures have been proposed to avoid this security mechanism integrity problem, they either involve the application of non-standard hardware or rely on host-bound application programming interfaces (API). This is why, so far, they are only used in the field of network intrusion detection.

In this paper, we present our results concerning a concurrent host memory sampling mechanism based on direct memory access (DMA) and demonstrate that it is possible to de-couple GPU kernel execution, thereby providing temporary isolation from the host and allowing data sampling actions to be taken without interruption. We present a security analysis of our approach and detail a proof of concept implementation of the autonomous concurrent monitoring and sampling system, thus, validating that self-sufficient data sampling using a commodity coprocessor (i.e. a GPU) is indeed possible.

7.1 Introduction

The reliable detection of attacks, particularly subversion attempts, often requires sensor placement on the host. Even moderately sophisticated attackers will however target host-based audit data collection and security controls (i.e. preventive and detective safeguards) as one of the primary stages of any attack. As a result, a successful subversion attempt will invalidate audit data and have the ability to obfuscate corresponding evidence after a very short interval. Both detection and possible mitigation actions therefore depend on identifying an attack within a small window of opportunity and must be sufficiently trustworthy to ensure that they have not themselves become the initial target of an adversary.

While a number of mechanisms have been proposed to achieve this objective (cf. Section 7.2 for a brief summary), these often impose additional requirements on the hardware and software architecture (cf. Petroni et al. [15] using a single-board computer on a PCI add-in card).

Furthermore, detection mechanisms operating on the host under observation itself generally suffer from two intrinsic flaws of such a host/observer combination. First, they are fairly easy to attack, since any software running *next* to the detection mechanism is susceptible to rather simple exploitations and can in turn be used to subvert the observer as such. Second, their results are vulnerable to straightforward man-in-the-middle attacks as the communication infrastructure and services used for this purpose are also host-dependent.

Detection mechanisms executed by coprocessors can generally perform limited tasks at a higher speed and, can also benefit from self-protection, as presented by [13]. The gain in speed is due to a smaller instruction set, a special (often parallel) architecture, and lower overhead. With respect to the gain in security, a narrow-enough programming interface between such a peripheral device and the host can already reduce the number of possible attack vectors, and can thereby intrinsically safeguard any software executed on it.

In this paper, we propose the use of a commodity coprocessor (i.e. the GPU) in order to operate a data sampling mechanism concurrently and asynchronously to the host. This insulates the data sampling and removes any further overhead from the host system. By leaving the standard GPU programming model (cf. Section 7.5) aside and further utilizing the fact that GPU subsystems are provided with a fast DMA pathway to the host's main memory, we developed a mechanism which allows the execution of code on a GPU without the host being actively involved.

By presenting a proof of concept implementation of an autonomous kernel, able to operate solely on the GPU while having unrestricted access to the host's physical memory, we validate that self-sufficient data sampling is indeed possible. The scope of such physical data observation leaves aside OS-level abstraction and is therefore suitable for, e.g., the observation of critical invariants as can be found in function tables. By outsourcing a self-sufficient data sampling mechanism to a commodity coprocessor, we fulfilled a condition for more reliable attack detection. With an autonomously operating off-host observation and data acquisition component, we counter drawbacks of current coprocessor approaches and make progress towards a concurrent detection of host event signatures using GPU coprocessing.

The remainder of this paper is structured as follows: We briefly review related work for concurrent realizations of security architectures in Section 7.2 before detailing our own ideas in Section 7.3. A security analysis is presented in Section 7.4, followed by an explanation of the key to operating GPU kernels autonomously and our mechanism to achieve this in Section 7.5. We close this work with conclusions and an overview of ongoing and future work in Section 7.6.

7.2 Related Work

The problem in providing a trustworthy monitoring mechanism for host security is closely linked to the need to ensure complete isolation of the security mechanism from untrustworthy processes, as found in the reference monitor design principles [2]. As the achievable assurance is, however, often limited by the scope of functional requirements, approaches such as the *separation kernel* proposed by Rushby [17] have attempted to provide such isolation through a suitable combination of hardware and software. With respect to current trusted computing group (TCG) architectures, Bratus et al. [3] make clear that trusted platform modules (TPM) only guarantee system integrity at load-time, leaving the detection of run-time vulnerabilities to additional mechanisms.

The application of parallel coprocessors for intrusion detection purposes is currently limited to the network level. Here, the pioneering works of Jacob and Brodley [11] and Vasiliadis et al. [23] are those with the most practical relevance. The former implemented a prototype called PixelSnort which ported the string-matching portion of SNORT, an "open source network intrusion prevention and detection system,"[1] onto a GPU, outperforming the traditional SNORT by 40%. In contrast to the present paper, Jacob and Brodley did not address the performance impact data sampling can have, and report similar CPU overhead for both SNORT and PixelSnort. Vasiliadis et al. took advantage of NVIDIA's compute unified device architecture (CUDA) technology and showed the potentials of existing pattern and string matching algorithms when ported to a modern GPU by using the corresponding runtime and development framework. Their prototype, called GNORT, achieved an improvement of data throughput of 100% with respect to the traditional SNORT system. A performance improvement to the GPU version of the Aho-Corasick algorithm was presented in 2010 by Tumeo et al. [22]. By taking into account the architectural constraints of a modern NVIDIA GPU, a speed-up of 6.67 compared to the original CPU implementation found in SNORT was achieved.

[1] www.snort.org

Huang et al. [10] propose a GPU-based multi-pattern matching algorithm derived from the idea of Wu-Manber [26], and report the execution performance as compared to the actual Wu-Manber algorithm, using the open graphics library (GL) API to control the communication between CPU and GPU. While using host APIs may be an option when performing network intrusion detection, it poses risks on the reliance of a detection mechanism when applied in the field of host intrusion detection.

In 2004, Petroni et al. [15] presented a proof of concept implementation of a coprocessor-based kernel runtime integrity monitor called Copilot. They used a single-board computer on a PCI add-in card in order to establish a link between the host to be observed (i.e. target) and the observing host. The lowest performance penalty on the system under observation is reported to be less than 1%, applying a sampling interval of 30 seconds. Given such an interval, rootkits tampering with process tables in particular may not be detected in time.

Besides Williams and Spafford [25], who presented a coprocessing intrusion detection system called CuPIDS that used a dedicated symmetric multi-processor (SMP) rather than a real coprocessor in 2007, the current research in the field of coprocessors for intrusion detection either concerns itself with the use of GPUs for assessing network traffic or proposes the application of uncommon hardware for host intrusion detection.

With regard to this, the use of graphics chipsets [13] and GPUs in general (i.e. commodity coprocessors) [19] is the subject of our previous work and was presented in 2010. While the former is also concerned with the self-protection of the coprocessor itself, the latter introduces a cost model capable of expressing the performance impact of observations in concurrent non-uniform memory architectures (NUMA). The practical validation of this model was reported in the same year and is the subject of [20]. The constraints with regard to modern GPUs and their off-the-shelf feasibility for use in asynchronous observations and host intrusion detection are the subject of [16].

Besides their use for intrusion detection purposes, GPUs have been applied in various other security-related domains. For instance, Cook et al. [4] and Harrison and Waldron [9] implemented symmetric key ciphers on the GPU. Both authors could not, however, report a performance advantage in favor of the GPU over the corresponding CPU implementation, and criticize a high CPU utilization partly caused by the runtime frameworks. Additionally, modern GPUs are used in the field of digital forensics. Here, Marziale et al. [12] presented an approach in which the GPU was taked with file carving (i.e., finding files by a set of predefined header and footer definitions), an reported an incrase in performance in all test scenarios.

7.3 Details

As already briefly mentioned in the two previous sections, current data sampling approaches show at least one of the following two shortcomings: Strong host dependence and use of non-standard hardware.

The present work does not claim to present a finalized general data sampling approach, but rather to form the basis for future work geared towards a novel and fully-fledged system. This ground work is important, as it is the common position that a kernel "(...) cannot be initiated as a stand-alone application, and strongly depends on the process that invokes it." [24]. This is true for cases where the GPU is configured and programmed using host-side APIs, which makes sense in many non-security and even non-host security-related use cases.

In order for a data sampling mechanism to be as tamper resistant as possible, a high level of independence from the host is inevitable. In cases where the mechanism is implemented within the very same host system it is supposed to protect, autarky is non-existant, whereas solutions employing auxiliary hardware can take advantage of a very high level of independence if used correctly. As host security, in particular, needs to be convenient and effective at the same time, approaches in which non-standard hardware is involved

(cf. [15]) are problematic. Therefore, using commodity coprocessors such as the GPU in order to fulfill the afore-mentioned twin objectives seems promising. For instance, a detection mechanism running concurrently on a coprocessor is able to reliably observe the host and to autonomously process the data in question without the need to wait for a time-slice to be assigned to it by the host under attack. In contrast to this, intrusion detection mechanisms running on the host itself can easily be exploited through host OS vulnerabilities, even when assigned to a dedicated CPU processor (cf. [25]).

How data of active processes can be retrieved by a coprocessor and the degree of performance degradation on the part of the affected processes has been subject of previous work (cf. [20]).

Since we circumvented the technical restrictions described in [16], concurrent and autonomous host data sampling using modern GPUs is now possible and permits continuous observation of critical data structures.

In order to keep the level of independence of a sampling mechanism as high as possible, logical host connections must be avoided. To this extent, configuring and programming the device tasked with executing the mechanism in its native language is necessary. As practically shown by Petroni et al. [15] and documented by Smith [21], coprocessors can fulfill this requirement but usually come along as non-standard peripheral hardware. In contras, the present work proposes the use of modern standard GPUs for executing a sampling mechanism in an autonomous and concurrent manner. Compared to other coprocessors, which are either not commodity hardware (e.g. field programmable gate array: FPGA), or are limited in terms of programmability and applicability (e.g. arithmetic logic unit: ALU), modern GPUs are considered to be standard hardware, computationally powerful, and programmable beyond their graphics scope.

We take advantage of the GPU's capability to communicate with the host system's memory by applying the direct memory access technology, which avoids any abstraction from layers such as the host's operating system. The sampling logic functions continuously and in total ignorance of the CPU utilization, as it is started as non-preemptive GPU kernel, being executed by dedicated shaders. This allows the commonly workload-lacking GPU to pursue its normal duty in parallel. Implementing the sampling mechanism as a non-preemptive kernel makes a controlling service obsolete, thus, reducing overhead and eliminating the need for logical host connections, while consuming a predefined maximum amount of GPU processing power.

The actual challenge of this approach comes from the fact that the even modern GPUs are not developed to be used as independent auditors, even though they fulfill a range of properties to operate as such (cf. [16]). While the application of GPUs for non-graphics computations is common in non-security and even non-host-security environments, the degree of independence needed for a GPU to be useful in host-related use cases cannot be achieved when using current programming frameworks. Nevertheless, enabling such a high-performance and almost omnipresent coprocessor to perform host-security related operations can be highly profitable. In the present work, we focus on two aspects: First, we analyze the security of our approach based on known attacks against GPUs and the techniques used to access and copy host data in Section 7.4. Second, we present the technical details of a proof of concept implementation of a concurrent and autonomous GPU kernel in Section 7.5.

7.4 Security Analysis

Our security mechanism, i.e., the sampling mechanism presented in Section 7.5, is executed concurrently and asynchronously on a GPU.

We are aware of the general problem of bootstrapping trust in commodity computers. This problem is thoroughly discussed by Parno et al. [14]. The authors present a survey of related work covering existing approaches and techniques to validate the integrity

of local and remote computers. This includes "mechanisms for securely collecting and storing information about the execution environment". The computation of cryptographic hashes over software binaries including all referenced libraries at a point in time where the software has not yet booted up is one example. As the trust put into the CPU is mostly very limited, additional hardware is often used to secure an information system's startup. Commonly, on-board modules such as trusted platform modules are applied to approve a system's integrity, despite the fact that these modules are intrinsically not as immune to physical attacks as, for instance, secure coprocessors such as the IBM 4758. The fact that even BIOSs are threatened has recently been brought up again by Symantec researcher Ge [6] who details the functionality of a virus known as "Trojan.Mebromi that can add malicious components into Award BIOS". While the task of reflashing a GPU's BIOS is similar to reflashing the system BIOS, compromising the proposed observation mechanism *from the inside* seems possible, but, to the best of our knowledge, such an attack is yet to be validated.

Since GPU software (e.g. driver, BIOS, etc.) is complex and tuned for performance rather than security, the inference can be made that it contains undiscovered vulnerabilities. As soon as GPUs are within the focus of cyber criminals, software vendors will have to react accordingly. A prominent example showing that this threat is real is a browser exploit based on the WebGL (web graphics library) technology. When applied, frozen desktops (OSX), full system crashes (Windows XP) and reset GPUs (Windows 7) are known consequences [5]. The current workaround to this denial of service attack is the deactivation of WebGL. With respect to the proposed security mechanism, a first solution could be to operate it on a separate GPU, which would also be a good idea in order to guarantee maximum performance for both graphics computations and host observations.

In general, any off-host device (or mechanism in the broadest sense) profits from the fact that host vulnerabilities (e.g. host OS vulnerabilities) have only a very limited effect on them. For instance, the GPU runs properly even on infected host systems as it has its own basic input/output system (BIOS), a firmware, several processors, and physical memory, and thus does not rely on the CPU. Nevertheless, possible attacks against such a device exist and are subject to the remainder of this section.

In 2007, Rutkowska [18] presented three possible attacks against physical memory acquisition for forensic analysis by altering the address mapping scheme for PCI devices. To achieve this, an additional entry is added to the corresponding northbridge memory map, redirecting any request from the respective PCI device. Since the GPU is also connected to the northbridge, a similar attack against the proposed sampling mechanism might be possible.

Any signal between host and off-host coprocessor must inevitably pass several controllers which increases the risk of sophisticated man-in-the-middle attacks. If such a controller was to be re-flashed during a subversion attempt aimed at serving the invader as an additional pathway to the GPU's memory system, eavesdropping on the detection mechanism or even communicating with it would be possible. To this extent, Gonzalez [8] presented a customized PCI Ethernet card, transparently eavesdropping any packet sent or received on the target machine.

Instead of redirecting accesses, it may also be possible for an attacker to slow down the GPU to a degree that allows him to finalize his attack before detection. This could be achieved by synthetically generating traffic on the system bus used or directly on the GPU.

Instead of loading data onto the GPU's RAM, an attacker could also load the GPU's RAM into the host system. As modern GPUs come with large amounts of (often unused) memory, it is possible to use the graphics card's memory as swap space for the host [7]. Documented for a benign application, this technique could possibly be used to run a denial-of-service attack against our detection mechanism, in such a sense that the attacker maps the full device RAM into the host. As this would also cause the system to freeze, the attacker would have to find the trade-off point where the GPU is still able to perform its

normal duty while unable to perform security tasks at reasonable speed or at all.

The above shows, that countering off-host intrusion detection is theoretically possible, emphasizing the need for self-protection. As designing a tamper-proof computing environment is the ultimate goal, the present paper describes an off-host approach, able to eliminate obvious and straightforward attack vectors while at the same time reacting more quickly to malicious activities while they are ongoing.

7.5 Autonomous Sampling using a Commodity Coprocessor

We will now present our proof of concept implementation of an autonomously running kernel, or, in other words, a kernel that keeps executing even after the initializing host application is terminated. The implementation shows that properly configured kernels are able to read data from arbitrary host memory locations and are able to perform simple calculations on this data.

Since most of the limitations revealed by [16] are not intrinsic to GPUs but to their runtime frameworks, we circumvented the major disabilities by waiving of any such framework and implemented our application using an intermediate language (IL) provided by AMD called AMD IL [1]. For preparing the device, we applied tools provided by the Stream SDK[2].

Figure 7.1 shows a sequence diagram of our implementation. As current GPU frameworks cannot be intuitively utilized for building applications that act as autonomous host auditors due to the constraints identified by [16], prerequisites such as device initialization or context creation are done manually in our implementation. For this purpose, we developed a C++ client application called *Host Init* that takes care of such things.

For steps (1)-(6), we make use of the compute abstraction layer (CAL) in order to tune the device for our purposes and check that all parameters are adjusted to our needs. This includes, for instance, compiling the actual kernel (cf. Listing 7.1 for an extract) into an object specific to the interfaces provided and the device used. In order to validate that the kernel operates even after Host Init has terminated, we allocated some shared memory which is accessible by all components of our proof of concept. As our host was running a Linux distribution, we took advantage of the `shm_open()` command in order to open a

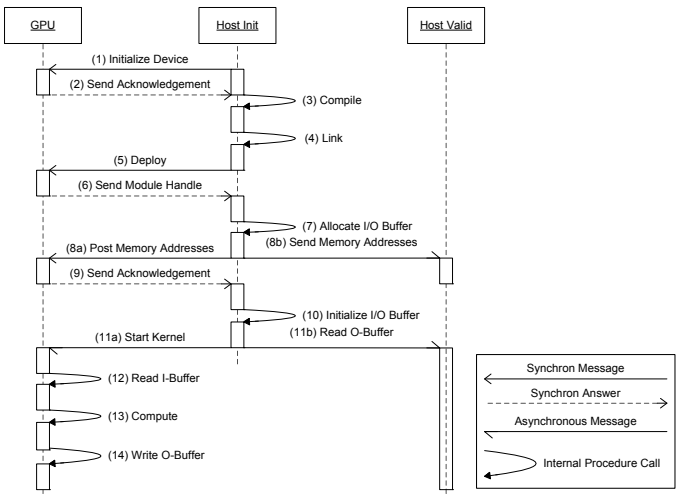


Figure 7.1: The sequence diagram of the proof of concept implementation of our asynchronous and concurrent sampling mechanism.

[2] www.amd.com/stream

shared memory object which can be operated on by the GPU, Host Init, and Host Valid similarly. Inside this object, two buffers (i.e. two two-dimensional arrays) were created: one for input and one for output. Once we have completed step (7), the addresses of both buffers are posted to the GPU in step (8a). Additionally, these addresses are sent to another host application called *Host Valid* in step (8b). Once this is done after step (9), we initialize the buffers in step (10). The handle received in step (6) is now used to start the kernel in step (11a). It is important to know, as is implied by the broken arrow, this is done asynchronously. At this point in time, Host Init is about to exit. Before it does so, it triggers the Host Valid application to start reading from the output buffer in step (11b). Each GPU thread executing the kernel repeats step (12) and (13) several times in a row before writing the result to the output buffer in step (14). Host Valid's only purpose is to validate that our kernel is functioning correctly, even though its initializing component has terminated, which prooves that our kernel is running entirely on the GPU and without the use of any host-side API.

We now present a shortened version of the source code of our kernel. The full kernel code can be requested from the author. Listing 7.1 presents a simple `while-loop` taken from our autonomous kernel, representing reading from, calculating, and writing to different buffers. It corresponds to steps (12) through (14) of Figure 7.1 and, in fact, serves as a symbolic placeholder for a matching algorithm.

Listing 7.1:
Extract form the GPU kernel written in AMD IL.

```
1 whileloop
2   ge r18.x___, r16.x, l0.x
3   break_logicalnz r18.x
4   mov r12.x___, v0.x
5   iadd r12.x___, r12.x, l1.x
6   iadd r16.x___, r16.x, l1.x
7 endloop
8 mov o0, r12.x
```

The calculations within the kernel are performed several times in a row before the result is written to the output. By assuring a longer runtime of the kernel, we give Host Init enough time to exit before Host Valid signals a change of the output buffer's value, proving that the kernel was running while Host Init was already terminated. These eight lines of code are executed independently from the host and validate that properly configured kernels can be used to perform an off-host host observation. While our implementation shows the general feasibility of such a task, enriching our kernel to the extent at which it can actually perform security-related tasks is just a matter of good programming skills. First performance tests revealed a maximum transfer rate between our kernel and the allocated

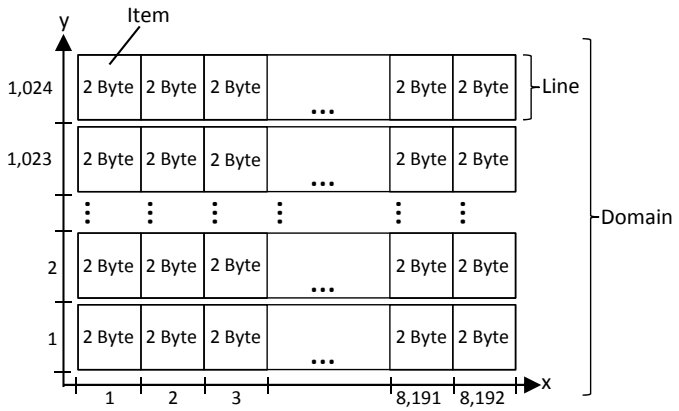


Figure 7.2: Logical memory architecture of our shared memory object.

shared memory of about 1GB/sec. In our scenario, we used an Intel Core 2 Quad (Q8200, 2.33GHz) host system with 4 GB of physical memory and a Club^{3D} HD4650 graphics card with 512MB GDDR2[3] memory. We utilized the full resource width of 8,192 items, where each item contained 2 Bytes, bound by x and y coordinates. Thus, one line contained 16KB. By configuring the kernel function to address a batch of 1,024 lines, we kept the overhead as low as possible, and were able to copy a domain of 16MB at a time. Figure 7.2 depicts the logical memory architecture.

Not taking advantage of the full resource width causes a higher overhead, which naturally results in a lower transfer rate. For instance, 30MB/sec. were achieved when reducing the domain to 256Byte (i.e. resource width of 128 and only one line).

7.6 Conclusion and Future Work

While commercial host intrusion detection systems are still being installed and executed on the host under observation itself, we propose a detection mechanism utilizing current commodity off-the-shelf hardware. With concurrency and asynchronism being two features of a modern GPU, in the current work, we presented a sampling mechanism to operate on such hardware. To this extent, in Section 7.5 a proof of concept of an autonomous running kernel able to access arbitrary memory addresses within the host's physical memory was detailed.

In Section 7.4, a security analysis was presented, showing that even an off-host host intrusion detection system needs to be capable of performing self-protection in order to succeed. We note, however, that the focus of the present paper is not on optimal general intrusion detection, and are conducting further work for both signature and anomaly detection in this framework beyond the scope of the present paper. Furthermore, in addition to increasing the sophistication of the observation mechanism in order to be used in real-life scenarios, its startup integrity and runtime security are within the focus of our future work. This includes assessing existing threats as well as designing new ones (cf. Section 7.4) in combination with possible remediation approaches.

Acknowledgement

The authors would like to thank Julian Knauer and Pierre Schnarz for their excellent practical support.

Bibliography

[1] ADVANCED MICRO DEVICES. AMD Intermediate Language (IL): Reference Guide, July 2011. (Last checked: August 22, 2012). 96

[2] ANDERSON, J. P. Computer Security Technology Planning Study. Tech. Rep. ESD-TR-73-51, Vol. II, James P. Anderson Company, Fort Washington, PA, USA, Oct. 1972. (Last checked: August 22, 2012). 92

[3] BRATUS, S., D'CUNHA, N., SPARKS, E., AND SMITH, S. W. TOCTOU, Traps, and Trusted Computing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)* (Villach, Austria, Mar. 2008), vol. 4968 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 14–32. doi:10.1007/978-3-540-68979-9_2. 92

[3] Graphics Double Data Rate V.2

[4] COOK, D. L., IOANNIDIS, J., KEROMYTIS, A. D., AND LUCK, J. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *Proceedings of the the Cryptographers' Track at the RSA Conference (CT-RSA 2005)* (San Francisco, CA, USA, Feb. 2005), A. Menezes, Ed., vol. 3376 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 334–350. doi:10.1007/978-3-540-30574-3_23. 22, 93

[5] FORSHAW, J. WebGL – A New Dimension for Browser Exploitation, May 2011. (Last checked: August 22, 2012). 95

[6] GE, L. BIOS Threat is Showing up Again!, Sept. 2011. (Last checked: August 22, 2012). 95

[7] GENTOO. Using Graphics Card Memory as Swap, July 2010. (Last checked: August 22, 2012). 95

[8] GONZALEZ, G. M-ETH: Man in the Middle Ethernet, Nov. 2010. (Last checked: August 22, 2012). 95

[9] HARRISON, O., AND WALDRON, J. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)* (Vienna, Austria, Sept. 2007), P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 209–226. doi:10.1007/978-3-540-74735-2_15. 22, 93

[10] HUANG, N.-F., HUNG, H.-W., LAI, S.-H., CHU, Y.-M., AND TSAI, W.-Y. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINAW 2008)* (Okinawa, Japan, Mar. 2008), IEEE Computer Society, pp. 62–67. doi:10.1109/WAINA.2008.145. 28, 29, 93

[11] JACOB, N., AND BRODLEY, C. Offloading IDS Computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006)* (Miami Beach, FL, USA, Dec. 2006), IEEE Computer Society, pp. 371 –380. doi:10.1109/ACSAC.2006.35. 28, 92

[12] MARZIALE, L., III, G. G. R., AND ROUSSEV, V. Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. *Digital Investigation 4* (Sept. 2007), 73–81. doi:10.1016/j.diin.2007.06.014. 22, 93

[13] MCEVOY, T. R., AND WOLTHUSEN, S. D. Concurrent Host Integrity Protection and Intrusion Detection. Unpublished early draft, 2009. 44, 45, 54, 57, 67, 91, 93

[14] PARNO, B., MCCUNE, J. M., AND PERRIG, A. Bootstrapping Trust in Commodity Computers. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)* (Berkeley, CA, USA, May 2010), IEEE Computer Society, pp. 414–429. doi:10.1109/SP.2010.32. 30, 94

[15] PETRONI, JR., N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 2004), USENIX Association, pp. 179–194. (Last checked: August 22, 2012). 6, 8, 22, 24, 43, 58, 60, 62, 67, 91, 93, 94

[16] RIEDMÜLLER, R., SEEGER, M. M., WOLTHUSEN, S. D., BAIER, H., AND BUSCH, C. Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection. In *Proceedings of the 2nd International Workshop on Security and Communication Networks (IWSCN 2010)* (Karlstad, Sweden, May 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/IWSCN.2010.5497999. 6, 8, 9, 10, 12, 68, 78, 93, 94, 96

[17] RUSHBY, J. M. Design and Verification of Secure Systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP 1981)* (Pacific Grove, CA, USA, Dec. 1981), ACM, pp. 12–21. doi:10.1145/800216.806586. 92

[18] RUTKOWSKA, J. Beyond The CPU: Defeating Hardware Based RAM Acquisition, Feb. 2007. (Last checked: August 22, 2012). 58, 62, 69, 95

[19] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[20] SEEGER, M. M., WOLTHUSEN, S. D., BUSCH, C., AND BAIER, H. The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation. In *Proceedings of the 9th Annual Conference Information Security South Africa (ISSA 2010)* (Johannesburg, South Africa, Aug. 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/ISSA.2010.5588311. 5, 6, 9, 11, 12, 93, 94, 104

[21] SMITH, S. W. Secure Coprocessing Applications and Research Issues. Tech. Rep. LA-UR-96-2805, Los Alamos National Laboratory, Aug. 1996. (Last checked: August 22, 2012). 19, 94

[22] TUMEO, A., VILLA, O., SCIUTO, AND DONATELLA. Efficient Pattern Matching on GPUs for Intrusion Detection Systems. In *Proceedings of the 7th ACM International Conference on Computing Frontiers (FC 2010)* (Bertinoro, Italy, May 2010), ACM, pp. 87–88. doi:10.1145/1787275.1787296. 92

[23] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium of Recent Advances in Intrusion Detection (RAID 2008)* (Boston, MA, USA, Sept. 2008), vol. 5230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 116–134. doi:10.1007/978-3-540-87403-4_7. 28, 67, 92

[24] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. GPU-Assisted Malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)* (Nancy, France, Oct. 2010), IEEE Computer Society, pp. 1–6. doi:10.1109/MALWARE.2010.5665801. 29, 93

[25] WILLIAMS, P. D., AND SPAFFORD, E. H. CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection. *Computer Networks 51*, 5 (Apr. 2007), 1284–1298. doi:10.1016/j.comnet.2006.09.011. 30, 43, 67, 93, 94

[26] WU, S., AND MANBER, U. A Fast algorithm for Multi-Pattern Searching. Tech. rep., The University of Arizona, 1994. (Last checked: August 22, 2012). 28, 93

Chapter 8

A Model for Partially Asynchronous Observation of Malicious Behavior

Abstract

For a non-trivial attack to be successful in compromising a target, multiple causally related operations must be performed. Detecting such potentially unknown sequences is the core problem in intrusion detection.

In this paper, we focus on the problem of *observing* attacks over non-uniform, partially asynchronous event sets. We hence propose charactering attacks as partially ordered sets, and we show how these can be detected asynchronously, as will typically be the case in a modern computing architecture. By extending a naïve model incorporating subsets of known causal dependencies, enhanced observation strategies minimizing the number and cost of observations can be derived. This incorporation of knowledge regarding constraints on attack causality into observations allows for notable enhancements in the efficiency of detection. We also provide a simple example of an application of the model for the case of an intrusion detection system on a co-processor observing a host, although the model is intended for arbitrary non-uniform architectures and concurrent operations.

8.1 Introduction

The security of any host intrusion detection system (IDS) is limited by the ability of an adversary to compromise the IDS itself. The IDS must therefore be able to detect and mitigate attacks before itself becomes compromised. Recent advances in computer architecture with large numbers of (partially asynchronous) processing units have, moreover, made the conventional notion of an IDS questionable: While detection algorithms operate on a *snapshot* of observations, attacks may be ongoing in parallel to the detection and mitigation efforts.

Although it is possible to force synchronization of events (mainly memory access) in a host system, this is highly undesirable, raising the necessity of performing observations asynchronously or minimizing the need for them. Previous work has sought to characterize the effects of non-uniform memory architectures (NUMA) by such observations, but has not been concerned with characterizing the observations further. In this paper, we seek to describe a model of general attack characteristics which can be used to optimize observation strategy. To demonstrate this, we describe a naïve model of causal but equiprobable asynchronous events, observed by way of memory access forcing point-wise synchronization, and we expand this model into an explicit causality model where event sets are constituted from partially ordered sets. These sets are also assigned arbitrary probability distributions, demonstrating a significant gain in efficiency.

The remainder of this paper is structured as follows: Section 8.2 briefly reviews the related work, before we start with the actual contribution of this paper in Section 8.3, where we detail our naïve model and our causality model. In order to present the practical relevance, especially of the latter model, we show the application of both models in Section 8.4 before we end the paper with a conclusion and an outlook in terms of current and future work in Section 8.5.

8.2 Related Work

Lee et al. propose an algebraic model for describing causality interfaces between actors, capable of, for instance, revealing causality loops, given that each actor is defined using a Turing-complete language. Schwarz [12] focuses on causality in distributed computations and presents a corresponding notion which is applicable in distributed, asynchronous systems. While not aimed at information security, his work covers some of the most important properties needed by an observer in order to reliably obtain the status of a distributed computation. That is, for instance, taking into account that "*every cause precedes its effect*".

Lamport's *happened before* relation[1] [8], written as $\rightarrow$, provides an intuitive definition for the case of partial orderings in distributed (as well as in asynchronous) systems; this can also, however, be extended to total orderings. Even though Tarafdar and Garg [15] describe the happened before relation as "*the best possible approximation of real time order that one can make in a message-passing distributed system without external channels*", they also claim that it is the wrong model for expressing and describing potential causality. To this extent, they propose to use the potentially caused relation, written as $\overset{p}{\hookrightarrow}$.

A related modeling approach to the one proposed here was described for security protocols by Backes et al. [2]. It relies on a finite graph (referred to as a *causal graph*) to express causality between events. Whilst [2] is only applicable to cryptographic protocols, it demonstrates the utility of relying on causality as an abstraction when reasoning over security properties. While many IDSs incorporate fixed causal models, such as rule sets, graphs, or automata, the more general case has not been fully explored for this application. However, King et al. [6] described a causality-based intrusion detection system (CIDS) incorporating network- and host-based alerts, reasoning about their severity by interpreting the causal dependency between network events (i.e., packets) and host events (i.e., systems calls). As a result of this, CIDS is able to reduce the number of false-positives, giving each alarm more significance. Subsequent work [7] extended this approach in conjunction with existing IDSs to provide causal relations for suspicious host- and network-based states and events.

Ning et al. [10] propose the correlation of IDS alerts in order to construct attack scenarios. They tested their formal framework using the DARPA dataset from 2000 and report a significant reduction of the false alert rate. In contrast to [10], we do not rely on alerts generated by third-party applications as we focus on low-level data elements being the subject of an subversion attempt rather than services. Also, our definition of an attack (scenario) is more finely-grained (i.e., [10] correlate several *service attacks* while we correlate several *data element attacks*).

Alert correlation as such is the subject of Cuppens and Miège [5], and Benferhat et al. [3]. Both works focus on the reduction of alerts generated by an IDS. In this context, our work could be seen as the mechanism which generates such alerts, not based on malicious network traffic but on a set of critical host-based data elements.

The recognition of an intruder's intent is discussed by Cuppens et al. [4] and Wu et al. [16]. The former focus on the reduction of alerts presented to a network administrator, while we focus on the reduction of observation frequency as such, keeping the detection probability as high as possible. The latter apply the Dynamic Bayesian Network approach to model uncertainty, while we apply the noisy-or model to incorporate noise (i.e., uncertainty), and we focus on a manageable amount of host-side elements rather than large amounts of intrusive data.

Very recently, Albanese et al. [1] published their work presenting a scalable analysis of attack scenarios. They analyze network infrastructure in order to reveal the dependency of available services. Based on the importance and dependency of network components and services, a dependency graph is constructed which is used later on for constructing (probabilistic) attack graphs. In contrast to [1], who rely on known vulnerabilities (i.e.,

[1] Also used by [12].

known exploits) rendering it impossible to detect yet unknown attacks, we focus on loosely coupled data elements which may be attacked during even unknown subversion attempts. Furthermore, as our model does not focus on the analysis of vast amounts of data, we do not need algorithms that "can process between 25 and 30 thousands [sig!] alerts per second" [1].

8.3 Observing Partially Ordered Attack Sets

Section 8.3.1 shows a naïve model capturing the worst-case scenario in terms of observation frequency and performance degradation due to coprocessor-triggered host observations when studying modeling outcomes. In contrast to this, Section 8.3.2 presents an explicit causality model. Here, we apply partially tailored versions of well-known models in order to incorporate our assumptions. This results in only a moderate decrease of detection probability whilst at the same time reducing the performance impact.

We focus our attention on a finite $POASET$ (i.e., partially ordered attack sets) of shared host-side data elements. We assume an attacker needs to inevitably tamper with all $de_i \in POASET; i = 1, ..., n$ in order to complete his malicious intentions, which may either target the host system as such or the in previous publications (cf. [9] and [13]) proposed off-host host observation mechanism. Formally, we define $\alpha_j = \{de_i, ..., de_m\} \subseteq POASET; 1 \leq m \leq n$ to be attack sets, where each α_j consists of at least one de_i. No further assumptions other than partial ordering are made for the events constituting attacks.

In practice, the exact content of the $POASET$ depends on parameters such as observation goal (e.g. privilege escalation) and type of system. Despite this, in order to achieve his malicious goal, an attacker must tamper with the full $POASET$ and is thus confronted with the problem that some of the concerned data elements depend on others. Therefore, he must build attack sets which have to be tampered with in order. This fact highly limits an attacker's possibilities when choosing his course of action, and it leaves room for causality-based observations (cf. Section 8.3.2).

Table 8.1 details the different assumptions made for each model, with respect to the performance degradation caused on the host when applying it.

8.3.1 The Naïve Model

The naïve model makes no explicit assumptions regarding certain data elements and their likelihood of becoming the subject of an attack, nor does it assume any causal dependency between attack sets. It presents the worst-case scenario in terms of observation frequency and performance degradation and validates the assumption that modeling causal dependency between data elements is likely to reduce the loss of performance.

While the set of all attack sets $\mathcal{A} = \bigcup_{j=1}^{k} \alpha_j$ forms a partial order fulfilling the *happened before* relation [8], each α_j itself does not represent an order of any kind. The happened

Description of Assumption	Naïve Model	Causality Model
Uses the happened before relation $\rightarrow$	✓	✗
Uses the potentially caused relation $\overset{p}{\hookrightarrow}$	✗	✓
Causal dependency between attack sets	✗	✓
Uniform distribution of attack probabilities	✓	✗
Maximum detection probability	✓	✗
Maximum performance degradation on the host	✓	✗
High false-positives rate	✓	✗
Observation succeeds always	✓	✗
An observation attempt can be suppressed	✗	✓
Legitimate changes of value can happen	✗	✓

Table 8.1: Comparison of assumptions incorporated into each model.

before relation defines – among other – that if α_j is attacked before α_{j+1}: $\alpha_j \rightarrow \alpha_{j+1}$ and if $\alpha_j \rightarrow \alpha_{j+1}$ and $\alpha_{j+1} \rightarrow \alpha_{j+2}$: $\alpha_j \rightarrow \alpha_{j+2}$. Obviously, $\rightarrow$ is transitive.

In other words, an attacker may freely decide in which order he wants to alter the data elements *within* each attack set α_j. This gives him $\sum_{j=1}^{k} |\alpha_j|!$ possibilities to do so, presuming that he alters each data element only once. He must, however, finish the attack against α_j before he can attack α_{j+1}.

This confronts the observer with two extreme cases where $\alpha_1 \equiv POASET$ and where $|\alpha_j| = 1$. We deduce from the former that an attacker has $|POASET|!$ possibilities to complete his attack while the latter extreme represents full serialization. Since any attack will be composed of loosely coupled attack sets, and because of the constraint that these sets have to be executed in order, any detection mechanism faces the problem of the inability to be one step ahead of the attacker. That is, the full $POASET$ has to be observed according to the busy-wait approach and at maximum observation speed, which is not efficient due to the performance degradation this causes on the host-side [14]. This fact already implies that any reasonable assumption made with respect to a causal dependency between attack sets is likely to reduce the impact on the host-side, as target-oriented observations can be performed (cf. Section 8.3.2).

As shown before, an attacker can freely choose the order in which he alters the data elements within attack sets, while at the same time being restricted by the happened before relation with regard to the execution of data elements belonging to different attack sets. Once the attack sequence is determined, the probability that a specific de_i within the targeted α_j will be attacked (assuming uniform distribution) is:

$$P(de_i|\alpha_j) = \frac{1}{|\alpha_j|} \tag{8.1}$$

The existence of attack sets clearly limits an attacker's alternatives for achieving any malicious goal. If the number of attack sets and their composition would be visible to an observer, targeted and ordered observations would be possible. Unfortunately, the existence of these sets is not visible to anybody but the attacker himself. Without any further assumptions being made, an observer is therefore always confronted with the problem of observing *all* data elements of the $POASET$ with equal frequency. As no assumptions about causal dependency between attack sequences have been made, each data element is equally likely to be subject of an attack at point in time t_0; i.e., the starting point of an attack:

$$P(de_i|t_0) = \frac{1}{|POASET|} \tag{8.2}$$

We note that this is of particular interest for multiple concurrent scheduling entities (as is the case in multi-core, multi-processor architectures) and denote $AP = \{ap_1, ..., ap_N\}$ to be the set of entities (processors) used by the attacker. In the worst case, from a security point of view, an attacker can utilize all processors to capacity by assigning each processor a new data element to work with as soon as the work with the former data element has been finished. Following this schema, rather than having several processors alter one data element in parallel, limits the need of inter processor communication, reduces forced synchronizations, and is therefore faster.

For simplicity, we assume that each alteration targeted at each data element requires an equivalent amount of processor cycles and that attacking a data element costs as much as observing it. Hence, the abstract[2] *time* the attacker needs to accomplish his goal is given by the maximum number of data elements to be processed by a single processor:

$$t_{attack} = \left\lceil \frac{|POASET|}{|AP|} \right\rceil \tag{8.3}$$

[2]The concrete time depends on the actual processor speed.

As one data element cannot be split over several processors, the result is mapped to the smallest subsequent integer (i.e., ceiling).
Alterations of data elements can be detected if the observation mechanism is able to observe de_i before and after it has been attacked, and the data read during both observations differ[3]. While, due to our assumptions, an analogous equation accounts for the time needed to observe all data elements (i.e., $t_{observation}$), we are interested in gaining the maximum probability of detecting an ongoing attack. The probability to do so is 1 if we are able to observe the set of all data elements within the $POASET$ in less than the time the attacker needs to alter one data element $de_i \in POASET$. This correlation can be expressed by writing:

$$t_{observation} < \left\lceil \frac{t_{attack}}{|POASET|} \right\rceil \tag{8.4}$$

Clearly, Equation 8.4 is a simple model leaving aside any causal dependency between data elements and assuming uniform distribution of attack probabilities. Also, employing this model would cause a high false positive rate, as even legitimate changes to any $de_i \in POASET$ would be treated as an attack. Furthermore, since data elements cannot be spread over processors, altering one data element exclusively and altering up to $|AP|$ independent elements in parallel consumes the same amount of time.

With no further assumptions made, the likelihood of being attacked is the same for all data elements within the $POASET$. We have therefore shown that, for maximum detection probability, an observer mechanism must be much faster than the processors used for attack. This is neither a surprise nor a problem in reality: Coprocessors such as modern GPUs are able to operate much faster than modern CPUs. Hence, the problem is not the speed as such but the performance degradation high-speed bulk observations cause on the host-side.

We define the performance degradation D to be maximum (i.e., 1) if Equation 8.4 is fulfilled:

$$P(D = 1 | t_{observation} < \left\lceil \frac{t_{attack}}{|POASET|} \right\rceil) = 1 \tag{8.5}$$

Below, we present the causality model, incorporating resilient assumptions regarding the success probability of an observation, as well as possible causal dependencies between attack sets. This reduces performance degradation whilst maintaining acceptable detection probability.

8.3.2 The Causality Model

In the previous section, we have associated each data element $de_i \in POASET$ to have the same probability of being the *first* attack target (cf. Equation 8.2) . Also, we have assumed that an observer has no knowledge about the sequence according to which an attacker will aim at a specific de_i, nor that he knows anything about the correlation between different data elements.

For the causality model, we allow further assumptions to be made but limit our attention to causality chains, despite the fact that more complex relations between sets of data elements may be possible. As we do not know the exact course of any attack, our assumptions are probabilistic in type.

The discussion up to this point has tacitly assumed that costs are equal for each event observation. However, this is not desirable for a NUMA architecture with different memory pathways, where each of them possibly has a different priority. As one result, not every attempt to observe a data element is assumed to be successful. This is intrinsic to our coprocessor/host combination (i.e., a NUMA architecture) which imposes different priorities to concurrent memory accesses depending on the source of them.

[3]For simplicity, we assume that the observation results are not stored.

We distinguish between two sets of memory pathways, being H, the ones triggered by the host itself, and C, the ones triggered by a coprocessor. Each memory pathway $h_s \in H$ and $c_t \in C$, respectively, is associated with a priority (i.e., p_s and p_t). With respect to an operation on de_i, H and C form a partial order over the happened before relation, such that $p_s \geq p_t \Rightarrow h_s \rightarrow c_t$. Whenever this relation is true for any two competing memory pathways h_s and c_t, we regard the observation as failed. For simplicity, we postulate that the set of memory pathways follows a binomial distribution. Therefore, assuming host-triggered events to have a higher priority, the probability of a successful observation can be expressed as follows: Let $p = \frac{|C|}{|C \bigcup H|}$. Then:

$$\hat{x}_i = B(x|p,n) = \binom{n}{x} p^x (1-p)^{n-x} \tag{8.6}$$

where x denotes the number of possibly conflicting memory pathways, p, the ratio between coprocessor and host-triggered events targeting a certain data element, and n, the number of successive observation tries.

Thus, $\hat{x}_i$ gives us the probability that exactly x memory pathways can successfully observe data element de_i without being overridden by a host-triggered memory pathway. While parameter x serves as a lower bound, we are in fact interested in the probability that *at least* x memory pathways can be used unrestrictedly. This is obtained by applying Equation 8.6 to all values $w = x, ..., n$:

$$\hat{X}_i = \sum\nolimits_{w=x}^{n} \binom{n}{w} p^w (1-p)^{n-w} \tag{8.7}$$

The smaller $\hat{X}_i$ is, the lower the probability that we can detect a change in value at de_i while an attack is ongoing, as this is the probability that we can observe a given data element without being suppressed by memory pathways triggered by the host. We refer to $\hat{X}_i$ as noise (i.e., λ_i) and make use of the noisy-or model [11] in order to deal with the possibility that an observation attempt is not successful. In addition to another memory pathway suppressing our observation, we have to incorporate a second uncertainty: The probability that we observe a benign change in value at de_i which we cannot distinguish from a malicious one. With respect to the noisy-or model, we speak of this factor as a global confidence probability, denoted as λ_0 [4].

Since we are interested in the causal dependency between attack sets, we make use of the *potentially causes* relation $\overset{p}{\hookrightarrow}$, proposed by Tarafdar and Garg [15] who describe it as a relation capable of expressing the connection of two events not sharing a local clock while at the same time having the possibility of one causing the other. In our case, we say that O_j^m and O_{j+1}^m are two observations of malicious changes that fulfill the happened before relation (i.e., $O_j^m \rightarrow O_{j+1}^m$). Furthermore, if the changes observed have the potential to cause other data elements to change in value, then $\{O_j^m \rightarrow O_{j+1}^m\} \overset{p}{\hookrightarrow} O_{j+2}^m$ and if $\{O_j^m \rightarrow O_{j+1}^m\} \overset{p}{\hookrightarrow} O_{j+2}^m$ and $\{O_{j+2}^m \rightarrow O_{j+3}^m\} \overset{p}{\hookrightarrow} O_{j+4}^m$, then $\{O_j^m \rightarrow O_{j+1}^m\} \overset{p}{\hookrightarrow} O_{j+4}^m$. Just as in its original form, our version of the potentially caused relation is transitive.
The probability that a change in value of a subset of the $POASET$ is malicious can be written as:

$$P(O_j^m|de_i, ..., de_n) = 1 - ((1-\lambda_0) \prod\nolimits_{i=1}^{n} (1-\lambda_i)) \tag{8.8}$$

Here, O_j^m is a random variable with the probability that an observation of a change in $de_i, i = p, ..., n$ is malicious. λ_0 is the confidence probability, i.e., $1 - \lambda_0$ is the likelihood that a benign event causes a malicious change in value. λ_i is a noise parameter which is associated with the data being subject to observation. In cases where we have causal

[4] $\lambda_i \in [0;1];\ 0 \leq i \leq n$

dependency between data elements, we simply take the result from Equation 8.8 as a noise-parameter, define a global confidence probability, and apply the noise model as before for the purposes of this model.

$$\widehat{O_k^m} = P(O_k^m | O_j^m \rightarrow O_l^m) = 1 - ((1 - \lambda_0) \prod_{j=1}^{l} (1 - \lambda_j)) \tag{8.9}$$

Equation 8.9 implies that $\{O_j^m \rightarrow O_l^m\} \overset{p}{\hookrightarrow} \widehat{O_k^m}$ and gives us a probabilistic answer to the question of how likely it is to see $\widehat{O_k^m}$ after we have seen the attack sequence $O_j^m \rightarrow O_l^m$. With this probability, we can start a targeted observation of the data elements included in observation O_k^m and thus, have the ability to adjust the observation frequency for each set of data elements being observed according to the causalities assumed.

Clearly, prior to applying the causality model proposed above, a thorough analysis of concerned data elements, their relation to each other, and possible attack types they could be used for, is mandatory. Once this has been accomplished, the model is capable of giving the probability with which certain observation results cause another one. Thus, this model not only allows an observation mechanism to be tuned and therefore, to gain in efficiency, but also provides limited predictive ability.

8.4 Practical Application of Proposed Models

The following provides a highly simplified instantiation of the model proposed in the preceding section, again divided into the naïve and causality models. We assume our set of data elements comprises 12 elements: $POASET = \{de_1, ..., de_{12}\}$.

We also assume without loss of generality that an attacker has decided to perform an attack $\mathcal{A}$ having four attack sets $\alpha_{1,...,4}$ containing the following data elements:
$\alpha_1 = \{de_2, de_5, de_6, de_{10}, de_{12}\}$; $\alpha_2 = \{de_1, de_8\}$; $\alpha_3 = \{de_4, de_7, de_{11}\}$, and $\alpha_4 = \{de_3, de_9\}$.

8.4.1 Application of the Naïve Model

If the observation mechanism was to be aware of the above defined attack sets, the probability that, e.g., de_5 will be attacked is $P(de_5|\alpha_1) = \frac{1}{5}$ (cf. Equation 8.1). . Without further assumptions or information, we assume events to be equiprobable:
$P(de_i|t_0) = \frac{1}{12}, i \in |POASET|$ (cf. Equation 8.2)
Assuming no relation between all data elements within the $POASET$ and being confronted with an attacker employing 4 processors $AP = \{ap_1, ..., ap_4\}$, at least one processor needs to process $t_{attack} = \frac{12}{3} = 4$ data elements, which consumes the corresponding abstract time (cf. Equation 8.3). In order to achieve the maximum detection probability, an observation of the full $POASET$ has to be finished in less than the time the attacker needs to attack one single data element: $t_{observation} < \frac{4}{12} = \frac{1}{3}$ (cf. Equation 8.4). While achieving this would result in a detection probability of 1, presupposing that the $POASET$ is complete, it would also cause the maximum performance degradation on the host-side due to forced synchronisations: $P(D = 1|t_{observation} < \frac{1}{3}) = 1$ (cf. Equation 8.5)

8.4.2 Application of the Causality Model

We will now apply the probability model introduced in Section 8.3.2 in order to show that observing all $de_i \in POASET$ with equal frequency is not necessary. First, we define one host-triggered and one coprocessor-triggered set of memory pathways[5] for each of the 12 data elements used in the example. Next, we need the probability that at least two coprocessor triggered memory pathways (parameter x in Equation 8.6) can be employed subsequently. Ideally, the first two memory pathways selected (parameter n) are of this type, as this implies a failure rate f of only 1 (i.e., $\frac{n}{x}$). The more tries we need in order to select two

[5]For a better understanding, one can think of these pathways as sets of threads.

x	2	2	2
n	2	4	8
C_1	10		
H_1	5		
$\hat{x}_1$	0.44	0.30	0.02
$\hat{X}_1$	0.44	0.88	0.99
f_1	1	2	4

x	2	2	2
n	2	4	8
C_2	8		
H_2	4		
$\hat{x}_2$	0.33	0.30	0.02
$\hat{X}_2$	0.33	0.88	0.99
f_2	1	2	4

x	2	2	2
n	2	4	8
C_3	10		
H_3	10		
$\hat{x}_3$	0.25	0.38	0.11
$\hat{X}_3$	0.25	0.69	0.96
f_3	1	2	4

x	2	2	2
n	2	4	8
C_4	4		
H_4	8		
$\hat{x}_4$	0.11	0.30	0.27
$\hat{X}_4$	0.11	0.41	0.80
f_4	1	2	4

C_5	13		
H_5	12		
$\hat{x}_5$	0.27	0.37	0.09
$\hat{X}_5$	0.27	0.71	0.97
f_5	1	2	4

C_6	40		
H_6	6		
$\hat{x}_6$	0.76	0.08	0.00
$\hat{X}_6$	0.76	0.99	0.99
f_6	1	2	4

C_7	5		
H_7	10		
$\hat{x}_7$	0.11	0.30	0.27
$\hat{X}_7$	0.11	0.41	0.80
f_7	1	2	4

C_8	60		
H_8	7		
$\hat{x}_8$	0.80	0.05	0.00
$\hat{X}_8$	0.80	0.99	0.99
f_8	1	2	4

C_9	16		
H_9	2		
$\hat{x}_9$	0.79	0.06	0.00
$\hat{X}_9$	0.79	0.99	0.99
f_9	1	2	4

C_{10}	2		
H_{10}	18		
$\hat{x}_{10}$	0.01	0.05	0.15
$\hat{X}_{10}$	0.01	0.05	0.18
f_{10}	1	2	4

C_{11}	12		
H_{11}	23		
$\hat{x}_{11}$	0.12	0.30	0.27
$\hat{X}_{11}$	0.12	0.42	0.82
f_{11}	1	2	4

C_{12}	6		
H_{12}	40		
$\hat{x}_{12}$	0.02	0.08	0.21
$\hat{X}_{12}$	0.02	0.09	0.28
f_{12}	1	2	4

Table 8.2: Complete table of results when applying Equations 8.6 and 8.7 from Section 8.3.2 to the 12 sets of memory pathways from Section 8.4; one host and one coprocessor set for each of the 12 data elements.

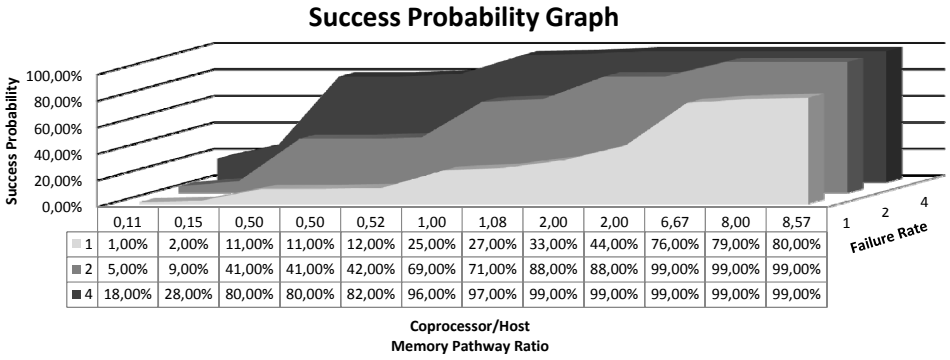


Figure 8.1: The results for $\hat{x}_i, \hat{X}_i$ and fr_i, $i = 1, ..., 12$

.

coprocessor triggered memory pathways, the higher the failure rate. Table 8.2 shows the results for $\hat{x}_i, \hat{X}_i$ and f_i, $i = 1, ..., 12$, where the success factor n is set to 2, 4 and 8, respectively. Table 8.2 reveals the connection between failure rate and success probability. That is, the more errors we allow, the higher the probability that we get two coprocessor triggered memory pathways. Correspondingly, minimizing the failure rate results in a lower success probability. The correlation between success probability, the coprocessor/host memory pathway ratio (i.e., $\frac{|C_i|}{|H_i|}$) and the failure rate is depicted in Figure 8.1. As we ordered the results according to the memory pathway ratio, it is clear to see that we can overcome the *failure rate problem* by assigning more coprocessor triggered memory pathways to each data element (i.e., success probability of up to 80% with a failure rate of 1 and a coprocessor/host memory pathway ratio of $8.57 : 1$). Given the success probabilities, we can now apply Equations 8.8 and 8.9 .

We then have to define our observation sets O_j, $j = 1, ..., g$. Ideally, we are not only able to define as many observation sets as the attacker has defined attack sets, but we also have them including the same data elements. While this may not be possible in general, we assume the following observation sets, gained from analyzing selected data elements: $O_1 = \{de_1, de_2, de_5, de_6, de_8, de_{10}, de_{12}\}, O_2 = \{de_4, de_7\}$, and $O_3 = \{de_3, de_9, de_{11}\}$. From our analysis we know that data elements de_3, de_9 and de_{11} (i.e., O_3) show a causal dependency to all other data elements. Therefore, we are interested in the strength of this connection. We start by calculating the probability that a change in value within O_1 and O_2 is malicious

(cf. Equation 8.8). We express the confidence in the preceding analysis by setting the global parameter λ_0 to be 70% for O_1^m, 60% for O_2^m, and 65% for O_3^m. Furthermore, we use the success probabilities from Table 8.2 (success factor $n = 2$) as λ_i (i.e., $\lambda_i \equiv \hat{X}_i$). This associates the probability of successfully performing an observation with the probability that the observation result is correct:

$$P(O_1^m|O_1) = 1 - ((1 - 0.70) \prod_{i \in O_1} (1 - \lambda_i)) = 0.99$$

$$P(O_2^m|O_2) = 1 - ((1 - 0.60) \prod_{i \in O_2} (1 - \lambda_i)) = 0.68$$

Since we are interested in the probability that O_1^m and O_2^m cause O_3^m (i.e., $\{O_1^m \rightarrow O_2^m\} \overset{p}{\hookrightarrow} O_3^m$) we now apply Equation 8.9 as follows:

$$\widehat{O_3^m} = P(O_3^m|O_1^m \rightarrow O_2^m) = 1 - ((1 - 0.65) \prod_{j \in O_1^m \bigcup O_2^m} (1 - \lambda_j)) = 0.99$$

This result is to be interpreted as follows: We have made resilient assumptions based on thorough analysis and thus, were able to form observation sets. We paid respect to the existing chance that our observation triggered by a coprocessor will not succeed due to host-triggered memory pathways of higher priority and subsequently used the resulting confident values as noise parameters of the noisy-or model.

Supposing the reasonability of our assumptions, we are 99% confident that a malicious change in value within the data elements covered by O_3 is potentially caused by malicious changes in value of the data elements observed by O_1 and O_2. In other words, once we have observed O_1^m and O_2^m, we can instantly lock O_3 and thus, will counter the subversion attack while it is ongoing with a probability of 99%.

With this result given, it is no longer necessary to observe all $de_i \in POASET$ with equal frequency which automatically influences the performance degradation positively. At the same time, there is a negligible loss in terms of detection rate (i.e., $\widehat{O_3^m} = 0.99$), compared to the busy-wait approach proposed in Section 8.3.1.

8.5 Conclusion and Future Work

We no longer have to be one step behind the attacker. Therefore, we are not interested in observing a full set of data elements without reasoning about what an attacker may do next. To this extent, we presented a model that focuses its observation on subsets of a partial order over host-side data elements. By associating each subset with a potential to cause the next action and applying two slightly tailored well-known models, we gain the following advantages: First, we are able to reduce the performance degradation on the host-side due to less interference. Second, we are able to jump at least one step ahead of the attacker, locking data elements likely to be attacked soon.

Our future work will be concentrated on implementing a proof of concept of the proposed models. We will focus on the access times of file system parts such as `/etc/passwd` and `/etc/shadow` and correlate the data with the frequency of, for instance, root logins. This first example is based on the assumption that after user specific data has been modified, a rising number of root logins will be the result.

Bibliography

[1] ALBANESE, M., JAJODIA, S., PUGLIESE, A., AND SUBRAHMANIAN, V. S. Scalable Analysis of Attack Scenarios. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS 2011)* (Leuven, Belgium, Sept. 2011), V. Atluri and C. Diaz, Eds., vol. 6879 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 416–433. doi:10.1007/978-3-642-23822-2_23. 102, 103

[2] BACKES, M., CORTESI, A., AND MAFFEI, M. Causality-based Abstraction of Multiplicity in Security Protocols. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF 2007)* (Venice, Italy, July 2007), IEEE Computer Society, pp. 355–369. doi:10.1109/CSF.2007.11. 102

[3] BENFERHAT, S., AUTREL, F., AND CUPPENS, F. Enhanced Correlation in an Intrusion Detection Process. In *Proceedings of the 2nd International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS 2003)* (St. Petersburg, Russia, Sept. 2003), V. Gorodetsky, L. Popyack, and V. Skormin, Eds., vol. 2776 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 157–170. doi:10.1007/978-3-540-45215-7_13. 102

[4] CUPPENS, F., AUTREL, F., MIÈGE, A., AND BENFERHAT, S. Recognizing Malicious Intention in an Intrusion Detection Process. In *Proceedings of the 2nd International Conference on Hybrid Intelligent Systems (HIS 2002)* (Santiago, Chile, Dec. 2002), IOS Press, pp. 806–817. (Last checked: August 22, 2012). 102

[5] CUPPENS, F., AND MIEGE, A. Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P 2002* (Oakland, CA, USA, May 2002), IEEE Computer Society, pp. 202–215. doi:10.1109/SECPRI.2002.1004372. 102

[6] KING, S. T., MAO, Z. M., , AND CHEN, P. M. CIDS: Causality Based Intrusion Detection System. Cse-tr-493-04, University of Michigan, Aug. 2004. (Last checked: August 22, 2012). 102

[7] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching Intrusion Alerts Through Multi-host Causality. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005)* (San Diego, CA, USA, Feb. 2005). (Last checked: August 22, 2012). 102

[8] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978), 558–565. doi:10.1145/359545.359563. 102, 103

[9] MCEVOY, T. R., AND WOLTHUSEN, S. D. Using Observations of Invariant Behavior to Detect Malicious Agency in Distributed Environments. In *Proceedings of IT Incident Management and IT Forensics (IMF 2008)* (Mannheim, Germany, Sept. 2008), vol. 140 of *Lecture Notes in Informatics*, GI, pp. 55–72. (Last checked: August 22, 2012). 53, 103

[10] NING, P., CUI, Y., AND REEVES, D. S. Constructing Attack Scenarios through Correlation of Intrusion Alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CSS 2002)* (Washington, DC, USA, Nov. 2002), ACM, pp. 245–254. doi:10.1145/586110.586144. 102

[11] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, 1st ed. Morgan Kaufmann, 1988. 106

[12] SCHWARZ, R. Causality in Distributed Systems. In *Proceedings of the 5th ACM SIGOPS European Workshop* (Mont Saint-Michel, France, Sept. 1992), ACM, pp. 1–5. doi:10.1145/506378.506423. 102

[13] SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34. 5, 6, 7, 8, 9, 11, 12, 27, 55, 62, 67, 68, 70, 71, 76, 77, 93, 103

[14] SEEGER, M. M., WOLTHUSEN, S. D., BUSCH, C., AND BAIER, H. The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation. In *Proceedings of the 9th Annual Conference Information Security South Africa (ISSA 2010)* (Johannesburg, South Africa, Aug. 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/ISSA.2010.5588311. 5, 6, 9, 11, 12, 93, 94, 104

[15] TARAFDAR, A., AND GARG, V. K. Happened Before is the Wrong Model for Potential Causality. Ece-pds-1998-006, University of Texas at Austin, July 1998. (Last checked: August 22, 2012). 7, 102, 106

[16] WU, P., SHUPING, Y., JUNHUA, C., AND ZHIGANG, W. Recognizing Intrusive Intention Based on Dynamic Bayesian Networks. In *Proceedings of the 1st Conference on Information Engineering and Electronic Commerce (IEEC 2009)* (Ternopil, Ukraine, May 2009), IEEE Computer Society, pp. 241–244. doi:10.1109/IEEC.2009.56. 102

Nomenclature

AES	Advanced Encryption Standard
AIDE	Advanced Intrusion Detection Environment
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
ASCC	Automatic Sequence Controlled Calculator
ASIC	Application Specific Integrated Circuit
ATI	ATI Technologies Inc.
ATPS	Adaptive Threat Prevention System
BIOS	Basic Input/Output System
BPSPM	Balanced Pattern Set Partition Method
CAD	Computer Aided Design
Cg	C for Graphics, a GPU programming language
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFA	Deterministic Finite Automaton
DLL	Dynamic Link Library
DpCAM	Decoded Partial Content-Addressable Memory
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
EDVAC	Electronic Discrete Variable Automatic Computer
ENIAC	Electronic Numerical Integrator And Computer
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
Gbit/s	Gigabit Per Second
GFlops	Giga Floating Point Operations Per Second

GHz	Gigahertz
GL	Graphics Library
GNU	GNU's Not Unix, a free software project.
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
IA-32	Intel Architecture 32-Bit
IBM	International Business Machines
IDS	Intrusion Detection System
IO	Input/Output
ISA	Instruction Set Architecture
kbit/s	Kilobit Per Second
KMP	Knuth-Morris-Pratt, a string matching algorithm.
LKM	Loadable Kernel Module
Mbit/s	Megabit Per Second
MHz	Megahertz
MIMD	Multiple Instruction, Multiple Data
NFA	Nondeterministic Finite Automaton
NIDS	Network Intrusion Detection System
NUMA	Non-Uniform Memory Architecture
OHCI	Open Host Controller Interface
OS	Operating System
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
Ph.D.	Philosophiae Doctor, Doctor of Philosophy
PHmem	Perfect Hashing Memory
RAM	Random Access Memory
SHA-1	Secure Hash Algorithm 1
SIMT	Single Instruction, Multiple Thread
SISD	Single Instruction, Single Data
SMP	Symmetric Multiprocessing
TPM	Trusted Platform Module
UDP	User Datagram Protocol
UMA	Uniform Memory Architecture
x86	A Processor Instruction Architecture

List of Publications

2012

- SEEGER, M. M., AND WOLTHUSEN, S. D. A Model for Partially Asynchronous Observation of Malicious Behavior. In *Proceedings of the 11th Annual Conference Information Security South Africa (ISSA 2012)* (Johannesburg, South Africa, Aug. 2012), IEEE Computer Society.

- SEEGER, M. M., AND WOLTHUSEN, S. D. Towards Concurrent Data Sampling using GPU Coprocessing. In *Proceedings of the 6th International Workshop on Secure Software Engineering (SecSE 2012)* (Prague, Czech Republic, Aug. 2012), IEEE Computer Society.

- SCHNARZ, P. UND SEEGER, M. M. Bürgerbefragung zur IT-Sicherheit im Endanwenderbereich. In *Datenschutz und Datensicherheit – DuD 36*, 4 (April 2012), Vieweg Verlag, 253–257. doi:10.1007/s11623-012-0094-6.

2011

- NAZAR, A., SEEGER, M. M., AND BAIER, H. Rooting Android – Extending the ADB by an Auto-Connecting WiFi-Accessible Service. In *Proceedings of the Conference NordSec 2011* (Tallinn, Estonia, Oct. 2011), Lecture Notes in Computer Science, Springer-Verlag, pp. 189–204. doi:10.1007/978-3-642-29615-4_14.

- SEEGER, M. M., AND BOURS, P. How to Comprehensively Describe a Biometric Update Mechanisms for Keystroke Dynamics. In *Proceedings of the 3rd International Workshop on Security and Communication Networks (IWSCN 2011)* (Gjøvik, Norway, May 2011), IEEE Computer Society.

- SEEGER, M. M. Using Control-Flow Techniques in a Security Context – A Survey on Common Prototypes and their Common Weakness. In *Proceedings of the International Conference on Network Computing and Information Security (NCIS 2011)* (Guilin, China, May 2011), IEEE Computer Society, pp. 133–137. doi:10.1109/NCIS.2011.126.

- SEEGER, M. M. Internetkonnektivität als Indikator für wirtschaftliche Stärke – Das Internet in Vergangenheit und Gegenwart. *Wirtschaftsinformatik & Management*, 2/2011 (Feb. 2011), 46–50.

2010

- SEEGER, M. M., WOLTHUSEN, S. D., BUSCH, C., AND BAIER, H. The Cost of Observation for Intrusion Detection: Performance Impact of Concurrent Host Observation. In *Proceedings of the 9th Annual Conference Information Security South Africa (ISSA 2010)* (Johannesburg, South Africa, Aug. 2010), IEEE Computer Society, pp. 1–8. *Selected for the Best Paper Award*. doi:10.1109/ISSA.2010.5588311.

- KEMETMÜLLER, C., SEEGER, M. M., BAIER, H., AND BUSCH, C. Manipulating Mobile Devices with a private GSM Base Station – A Case Study. In *Proceedings of the 8th International Network Conference (INC 2010)* (Heidelberg, Germany, July 2010).
- SEEGER, M. M. Three Years Hacker Paragraph. *Datenschutz und Datensicherheit – DuD 34*, 7 (July 2010), Vieweg Verlag, 476–478. doi:10.1007/s11623-010-0133-0.
- RIEDMÜLLER, R., SEEGER, M. M., WOLTHUSEN, S. D., BAIER, H., AND BUSCH, C. Constraints on Autonomous Use of Standard GPU Components for Asynchronous Observations and Intrusion Detection. In *Proceedings of the 2nd International Workshop on Security and Communication Networks (IWSCN 2010)* (Karlstad, Sweden, May 2010), IEEE Computer Society, pp. 1–8. doi:10.1109/IWSCN.2010.5497999.
- SEEGER, M. M., AND WOLTHUSEN, S. D. Observation Mechanism and Cost Model for Tightly Coupled Asymmetric Concurrency. In *Proceedings of the 5th International Conference on Systems (ICONS 2010)* (Menuires, France, Apr. 2010), IEEE Computer Society, pp. 158–163. doi:10.1109/ICONS.2010.34.