Concurrent Memory Inspection for Intrusion Detection

Christos V. Tsopokis

Technical Report

RHUL–MA–2014– 13

01 September 2014

Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX,
United Kingdom

Christos V. Tsopokis
Student Number: 100753427

Royal Holloway - University of London
Information Security Group

MSc Project

Concurrent Memory Inspection for Intrusion Detection

Supervisor: Stephen D. Wolthusen

August 2013

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, an in accordance with these regulation I submit this project report as my own work.

Date: ____________________

Student's Signagure

____

Abstract

Data communication devices and services are of rising importance and vitally rely on intrusion detection infrastructure. Such systems are common target for cyber-criminals who employ novel attack vectors through exploitation and concealment methods. In this context we should include the evolution of hardware which makes multi-core/multi-processor systems rather ubiquitous.

Unmasking techniques based on specification (e.g. signature-matching) are not capable of leading us a step after the attackers. Furthermore, concurrent systems are characterized by a non-deterministic scheduling behaviour that nullifies their static properties, thus making statistical analysis and machine learning rather difficult. Consequently, modern intrusion detection systems face various logic and physical challenges.

To reveal this kind of activities, on-the-fly energetic data probing is necessary. This is because we care both for the dynamic attitude of the analysis and the detection speed. In addition such systems – be it behavioral or anomaly-based – should also tackle the well known problem of performance degradation on the target host.

This project seizes upon the idea of anomaly-based host intrusion detection on distributed systems, with respect to the aspects mentioned above. Our main purpose is to explore the workability of such schemes and give a descent overview of the capabilities and limitations faces in this field. This is done by investigating potential sensor models based both on localization and outsourcing. Along with every model we investigate what shortcomings are faced, the performance impact and we try to enumerate all the factors that strike a balance between security, performance and convinience.

We provide all the background needed for formulating legal states, concurrently extracting data and investigating their deviation from well-known trusted stati. This project works as a feasibility analysis of anomaly detectors on parallel systems and as such it focuses on providing an overview of the challenges and promises offered by such a system. The material provided is open to updates since it is affected by a number of factors that affect numerous assumptions.

Acknowledgements

I would firstly like to express my great thanks to Dr. Stephen D. Wolthusen, who accepted to supervise this project from the very beginning of the academic year. Through his guidance, I was introduced in a rather demanding and multi-dimentional research area with which I was not familiar before. Had it not been for his well directed supervision, it would have been extremely difficult to find all the necessary resources and understand rather complicated concepts involved in this MSc project. I can definately say that his overarching scrutiny of technical aspects have stigmatized the way I perceive the importance of discipline on the research field.

There have also been many other people who challenged my ideas and enriched my research but regrettably enough, I cannot acknowledge all of them by name. However, I feel the need to send my appreciation and great thanks to my friends Nikos Kossifidis and Apostolos Glenis. The first as Linux kernel developer and the second one as a hardware researcher, contributed in bearing in mind some critical aspects that connect the design principles with practical limitations. Their observations bridged some gaps that cannot be understood without the necessary experience.

Last but not least, I would far and foremost like to thank the most beloved persons of mine. Without the support of my family I would never be able to attend the MSc in Information Security at Royal Holloway and be always free to make my own decisions. I am also grateful to Elena, who is at the core of my feelings, for standing with patience next to me throughout this year. This project is dedicated to my parents, Vasileios and Aggeliki Tsopokis, as a minimum indication of gratitude and respect.

Christos V. Tsopokis

25-08-2013

Nomenclature

API	Application Programming Interface
CFG	Control Flow Graph
CPU	Central Processing Unit
DFA	Data Flow Analysis
DFG	Data Flow Graph
DMA	Direct Memory Access
DSMP	Distributed Shared-Memory Multiprocessors
GPU:	Graphic Processing Unit
GPGPU	General Purpose GPU
IDS:	Intrusion Detection System
IDT	Interrupt Descriptor Table
LKM	Linux Kernel Module
MAC	Mandatory Access Control
NUMA	Non-Uniform Memory Access
OS	Operating System
PCI(e)	Peripherac Component Interconnect (express)
RAM	Random Access Memory
SSMP	Symmetric Shared-Memory Multiprocessor
UMA	Uniform Memory Access
VM	Virtual Machine
VMM	Virtual Machine Monitor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Νοῦς ὁρᾷ καὶ νοῦς ἀκούει, τ' ἄλλα δὲ κωφά καὶ τυφλά.

Ἐπίχαρμος

Most Intrusion Detection Systems (IDS) are based on signature matching approaches in order to detect malicious software (malware). This means that for every "group" of malware a unique identifier should be generated, stored and compared to every component candidate for excecution. The advantages and limitations of this technique are very well documented and it is generally proved that malware authors can always be a step ahead by employing polymorphic or metamorphic attacks. This along with the indeterministic behavior of concurrent environments, which become rather ubiquitous, make it easy to bypass signatures of well known samples.

To bridge this gap, one could monitor the symptoms instead of searching for potential causes of abnormalities. This is doable because malware mainly works by performing amendments on the kernel of the Operating System (OS) to generate "permanent backdoor(s)" in the infected host.

This project aims to perform an **exploratory feasibility analysis** of this approach by examinig alternative IDSs based on **detecting anomalies** within the current state-of-art in terms of hardware architecture and software engineering desing. With respect to the time limitations faced, our main target is to present a complete overview of the challenges and capabilities promised by this method. This can work as a blueprint for more specialized research which involves a lot of resources in terms of equipment and research time. Generally speaking, this documentaion sets a baseline for the investigation and assessment of memory monitoring systems and kernel control flow analysis by attempting to answer to the next question:

> *How could parallel environments be modeled to perform memory inspection capable of detecting malicious agency with respect to the computational overhead added by the underlying operations?*

The goal of this MSc project is to present a decent description of the challenges faced in this field of research and make an effort to underscore the difficulties occuring during the design, implementation, operation and update of these intrusion detection mechanisms. The above declared question is composed of some subscopes categorizing our investigation and finely shaping our orientation.

RQ1: How can we model reliable and efficient concurrent host observation?

RQ2: How could we model kernel's flux and detect potential (non) control-flow violations?

RQ3: Could the stohastic building assumptions of a parallel sensor system become subtly fuzzy by a sophisticated attacker?

The rest of this project is organised as follows: Chapter 2 reviews related work, before we start with presentation of the background research in Chapter 3, where we define related terminology and illustrate the limitations behind our motivations. In order to present the practical aspect of our research, Chapter 4 provides detailed investigation of concurrent memory inspection formulas along with factors affecting the cost models, followed by a study upon kernel control flow analysis in Chapter 5. Chapter 6 goes through a case study including further examination of aspects related to fine-grained observation measurements before we end this documentation with dimensions towards challenging future work and the conclusions of our study in Chapters 7 and 8 respectively.

Chapter 2

Literature Review

In this chapter we provide a short review of the research related to this project. We will start presenting our motivations by introducing the most common problems faced by traditional malware analysis and intrusion detection techniques. We will also point the reasons for which the dominant cause-oriented tactic of signature matching can be further degraded in multi-processing systems. After presenting our motivations we will introduce how an alternative symptom-based detection system can be proved to be more effective along with the reasons for which we consider it to be an approach with a strategic depth in its design. Having a generic understanding of the problems arising within the field of both signature and anomaly based intrusion detection systems is critical and we will attempt to present the prons and cons of each method with respect to concurrent systems' architecture. Moreover, we are going to explain why this alternative design is capable of covering the existent gaps of traditional defence lines and give some first indications of the challenges faced in the classic race between malware developers and intrusion analysts in this context, to make the usefulness of the examined method more concrete. Finally, we will present our research questions which are later going to be analysed and examined in the main chapters of our documentation.

2.1 Motivation

The most common strategy for malware detection utilized by most scanners, like Anti-Viruses and Intrusion Detection Systems, is based on syntactic signatures uniquely identifying each malicious program [52] . These signatures are mainly structured by an abstract representation of either the Control Flow Graphs (CFG) [18, 25] or Data Flow Graphs (DFG) [13, 32]. In realistic scenarios this is closely related to **static analysis**. That means that the binary of the malware is examinded, not at run-time but statically, by utilizing pattern matching techniques requiring a database of *known samples*, which are compared to the signature of any program under examination. Binary analysis is a powerful methon providing useful information for the work flow of a binary or its behavior but it is very much constrained by its static nature. To make it more clear, the nature of this analysis is an approximation of the program's run-time behavior. This predictive approach is what generates the fallacies and pitfalls of static analysis.

First of all, static analysis is proven to be limited by a number of novel obfuscation methods such as polymorphism and metamorphism [81, section 7.5 & 7.6], each of which has the ability to diversify the syntactic flow of the malware in a manual or automated way respectively. This is actually because syntactic signatures are ignorant of the semantics of instructions [52] and any tranformation upon them is most probably capable of evading detection. ***Kruegel et al.*** (**2004**) introduced a detector aware of binary semantics essentually oriented to Data Flow Analysis (DFA). This is more a "white list" approach where the behavior of benign Linux Kernel Modules (LKMs) is distinguished from Kernel Level Rootkis semantically (i.e. what type of operations they perform on specific kernel space addresses). This, as well as the symbolic execution of the binary, was proved to be very effective against almost every well known sophisticated rootkit (100% of successful detection) in conjuction with normal kernel modules with the satisfying results (0% of false positives). However, ***Cavallaro et al.*** (**2008**) and ***Moser et al.*** (**2007**) explain the reasons for which this kind of novel detectors can be bypassed actually by perfoming pseudo-dynamic/run-time-dependant operations or control dependecies on constant values by making the exploration of all possible paths of execution almost impossible. Furthermore, ***Seeger*** (**2011**) describes another restriction coming from the inability of prowerful static analyzers, "*to react to changes in the executional flow caused by dynamically loaded code*" like run-time function hooks.

These points briefly describe the limitations posed to static analysis but it is worth mentioning the additional cost of the very primitive assumption of some well *known samples*. This prerequisite leads to what is well known as zero-day attacks (this is mostly used for the exploitation of not previously known application bugs but its use here is in a similar context). In absense of a signature, none of these measures can be effective because one member of the equation is missing. To produce the new signature identifier, malware analysts are forced to perform dynamic analysis by executing the malware [81, section 15.4.4]. Practically, the execution of the malware is what distinguishes dynamic from the static one. On the fly observation of malware behavior, like file change monitoring, network port or connection monitoring, is a very demanding task. Furthermore, malware developers have adopted fine-grained techniques to detect the environment of execution and prevent proper execution in debugging or emulated environments [81, section 6.2.9]. It hence becomes clear that dynamic analysis is actually utilized to trace the operation of malware and come up with a semantic identification providing a source of detection in any future recurrence. We argue that this cause-based approach, where the reason that might provoke systemic misbehavior is examined, is going to be even more relegated by the "next generation of malware" which will be able to take advantage of the capabilities provided by concurrency of execution.

A key factor for that is the rapid and in a large scale dissemination of concurrent systems. Synchronization issues arising in such systems, increase exponentially the difficulty of perfoming binary static analysis. ***Taylor*** (**1983**) describes the problem of static intra-procedural analysis sensitive to synchronization issues (known as *rendezvous*) for concurrent programs is $\mathcal{NP}$ hard and the above mentioned article does not take into account recursive procedures. Techniques and algorithms of static analysis that take into account constraints imposed by synchronization issues affecting the execution flow of the program in question, can minimize the possible execution paths and make more accurate predictions about the potential run-time behavior of the program. However, this task is hard enough and if not successfully completed there will most probably be reports of "anomalies which correspond to sequences of events which lie along unexecutable paths." [82]. Additionally, any *rendezvous* missing will result in omitted erroneous condition report. It is hence evident that the CFGs or DFGs produced will most probably depict fake approximations of the real time work flow of the procedures. Things become even more complicated when it comes to recursive processes, where there is the need for contextual information from the return to the original call site. If it is not possible then the return edges have to be propagated back to all the potential callback sites. This problem is known as *Context-Sensitive Synchronization-Sensitive Analysis* and *Ramalingam* (**2000**) proves its undecidability. Conclusively, it becomes clear that the context becomes much more complicated for static analysis in parallel environments.

As a result a different approach to the problem is needed and it should be capable of giving strong indications of malicious (or abnormal) activities taking place into the system. Obviously, this cannot be oriented to static methods but on the contrary there have to be dynamic estimations making conclusive remarks upon the running behavior of the target system. One could suggest the utilization of dynamic analysis by targeting the identification of malicious programs. However, this perspective will be obviously proved to be ineffective for three reasons:

1. it is still designed to act with a black list approach, i.e. still targeting to detect the cause of the problem but not the consequences of it,
2. there is still an amount of unlimited risk since it is does not exclude the scenario of a "0-day attack" as described above and
3. it requires the dedication of a great amount of computational sources that, as it will be explained later, could be proved to be much more efficient in a different way.

Having these characteristics in mind and the advanced computational capabilities provided by concurrent systems we will suggest an alternative direction for fighting malicious agency in these systems. To design such a strategy we should focus on the usual consistencies caused by operation of malware on a system. Generally, this is the approach known as **anomaly based intrusion detection**. Once a piece of malware is installed on a computer system it will be utilized to perform a number of actions like privilege escalation, new network connection establishment, information leakage (like password stealing), creation of new users etc. These kind of operations provide a number of advantages to attackers like remote control of the systems and long lasting existance of the malware in the system. In addition, the more sophisticated the malware is the more stealthy it should be. Classic methods to achieve this are the substitution of useful system applications or even more seriously subversion of the host kernel itself.

It thereafter becomes clear that the common factor of illicit activities is the manipulation of the infected system's kernel. The extend up to which these tamperings are detectable is under research but the current results are very encouraging. The fundamental idea behind this formulation is the sense of dedicating a number of (co)processors in a concurrent environment for shared memory inspection to detect content mismatches in specified kernel memory addresses. Of course, forming these "expected" and "unexpected" behaviors is not a trivial task but it is quite radical and promising as it provides a way of working on a number of finite legal states than on an infinite group of illegal indications. We restate that this is proved to be effective when parallel systems are in use. As we will se later on, the more computational power the host provides, the more effective the investigation can be. This appeals even more in cases of critical infrastructure protection where supercomputers of enormous capabilities are in use.

The foundamental necessity is the ability of a mechanism, once an attack takes place, to detect and report it. The integrity of the system state is what can only provide evidence about this fact and a fine-grained way to validate it is needed. A malware designed to act in such a way and operate in an already exploited environment to subvert the systems for reasons of sthealthiness is called **rootkit**. To the best of our knowledge, the first proposals for coprocessor utilization to detect systemic discrepancies were presented by ***Molina and Arbaugh*** (**2002**) and ***Zhang et al.*** (**2002**). One of the first ideas for internal side protection was the development of filesystem integrity checkers providing cryptographic mechanisms for integrity validations on crucial files of the system that are usually exposed to attacks. Instead of performing a mutual trust approach where the examined system is considered trustful, without depending on the integrity of the underlying system an external auditing mechanism exists and provides integrity detection checks of peripheral devices like hard disks where critical system files occur. This work is very interesting as it is modeled on the principles of autonomous supervisors, in this case through an external device accessing the host memory through a Direct Memory Access (DMA) method. Although, it is a major advantage that the security of the observer does not rely on the system under examination, the efficiency of additional hardware being required, forming an embedded subsystem (through a PCI bus), is under question.

Based on that, ***Petroni et al.*** (**2004**) present a similar mechanism named Copilot. The basic characteristic of this paper is that it implements integrity controls on static data of the kernel memory space like the system call table, the interrupt descriptor table (IDT) and the text segment. It also capable of dealing with unusual modifications of other data structures mostly changed by administrative actions by reporting the error and reperform the validation. However, Copilot is not capable of examining run-time modifiable and fully dynamic data structures and it is also vulnerable to a number of additional threats like *time attacks* and cache attacks. Whilst the scale up to which the latter can be considered a threat is not yet clear, the former is a factor of deep consideration which will definately attract our attention at the next chapters. The self protection of Copilot is based like previously in the isolation through an external channel of communication via a PCI card. Nonetheless, Copilot is a good guideline providing the necessary bases for building-up an extended model. Of course there is a number of limitations the main of which is the hypervision of stationary kernel regions, which is an issue of paramount importance for real and not just expiramental implementations.

Petroni et al. (**2006**) provide an extended auditing mechanism and in this case there is additional inspection for dynamic kernel data structures like the process account table and the access vector cache used by the SELinux Kernel Module. Detecting mismatches in data structures with dynamic behavior makes the model more complicated, by requiring a specification architecture sufficient enough to reveal suspicious states by observing intra-dependant data structures. Another important aspect of the research questions comming out from the same work is the prefered monitoring model i.e. synchronous versus asynchronous monitoring by presenting the challeges faced by each approach. In essense, by synchronous accesses the monitor has "consistent view of kernel data" but is more prone to attacks, whilst asynchronous accesses are more protected due to "isolation" but can obtain inconsistent memory snapshot for a number of benign reasons (e.g. memory accessing during the update of a kernel table). In parallel to that ***Loscocco et al.*** (**2007**) presented a similar module called Linux Kernel Integrity Monitor (LKIM) which is was also capable of inspecting critical kernel locations with dynamic behavior. The big difference is the existance of the module within the observed host in the kernel space at firstplace and in the user-space in secondhand to reduce the performance impact. This created an extra burden on the memory accessing through a more complicated interface, but paramountly exposed the monitor to absolute subversion.

Issues related to monitoring modeling and its related challenges which mainly are **self pro-**

tection and **false positives minimization**, will definately be part of our examination. A year later ***Baliga et al.*** (**2007**) argued that Copilot is effective in *previously known attack vectors* by introducing 4 different attacks that could not be detected mainly because they were attacking to kernel details not included in the original specification. At this point there is a fundamental question coming up. If we are again forced to fall into the race trap between attackers and defenders, why should we prefer this model? The answer is mentioned very clearly by ***Petroni et al.*** (**2006**) by saying "*Nevertheless, this approach is still better than the traditional signature-based virus-scanning approach in that each specification has the potential to detect an entire class of similar attacks, rather than only a single instance.*". Practically the level of defence is related to our understanding of the kernel and the completeness of the modeling we perform on it and every action taken to extend our protection mechanisms, drastically reduces the attack vectors that might probably arise. Apart from the novel attacks a comprehensive categorization of the attacks in general is provided and we consider it crucial to have well-defined partitioning of them since it can assist on designing generic defence mechanisms. ***Baliga et al.*** (**2011**) introduced a novel rootkit detector (called *Gibraltar*), which also works by utilizing an external PCI card monitor, alike Copilot, for memory capturing. However, this is a model with automated specification which overcomes problems posed by manual specification (e.g. the one implemented by ***Petroni et al.*** (**2006**), where specifications might not be supplied during configuration time).

The above mentioned introduce the motivating reasons for this project. In the next section we are going to provide a description of more specific research questions that we are going to work on during the next chapters by trying to examine fine-grained defined methods of monitor modeling.

2.2 Reseach Questions

Under the assumption that, the scalar nature of kernel data structures described above and the labyrinthine construction of it as a whole, we can obviously understand that the observation modeling is a very demanding procedure. By focusing on constantly systemic misbehavior caused by clandestine activity and the promising perfomance available in concurrent systems capable of revealing these anomalies we provide a more detailed overview of the research questions presented in the previous chapter by revising the results of the related publications.

2.2.1 RQ1: How can we model reliable and efficient concurrent host observation?

The complexity of the kernel in every operating system and the multiple possible transitions between the states of its dynamically modified data structures, requires the existance of a model flexible enough to represent all the possible changeovers. ***McEvoy and Wolthusen*** (**2008**) define such a design where the possible states are modeled as a partially ordered set with causal relations. Abstractly, each state corresponds to a node, forming a graph the edges of which are the state transitions accompanied by the assigned transitional probabilities. This approach is versatile enough to depict kernel states of various elements falling into different categories with either static or dynamic properties and the fact that its functionality is by default stochastic, provides the flexibility needed for this task. Furthermore, it is resistant to attacks and consequently the results of the observations are stressed by the "multiplicity, distinctness and independence of observers". Namely, the more observers we assign for the supervision of each critical data element or *semantic checkpoint* [47], the more difficult it is for an attacker to subvert the observations, since the whole group of observers has to be compromised. As a state of art it is preferable to previous self defence mechanisms based on an obscuring the sensor, in a virtual environment for example. As we will see later hardware-based isolation is safer for integrity reasons but there are restrictions rendering its ability to interleave inbetween the malicious activity. This trade off will definately be analysed in the next chapters but intuitively there are limitations in the precision of memory accesses making the probabilistic analysis of the results more complicated especially when it comes to intrusion data collection.

With respect to the payoff that a fully embedded surveillance mechanism within the observed host adds an extra risk since the intrusion detection system itself is exposed to host OS exploitation. As a result, the utilization of an autonomous coprocessor within the host system with great computational power would be a highly profitable and secure plan. Such components are the ubiquitious Graphic Processing Units (GPU) but there in a number of constraints for their use documented in detail by ***Riedmüller et al.*** (**2010**). Briefly, the most important ones are **(1)** limited access main memory of the host in the user-space, **(2)** operational dependency and delayed startup from the host system and **(3)** lack of intercommunication interfaces. GPU communicate with the hosts through DMA which provides a number of advantages as we will explain in consecutive chapters but certain attacks possible should be taken into consideration [68, 78]. Recent research by ***Seeger and Wolthusen*** (**2012**) investigates how these limitations can be overcomed with a proof of concept implemenation. We are going to examine later on the possible specimens of concurrent host observation. Enumerating the tradeoffs of each case might assist in finding an equilibrium point between them.

For one more time we see that such models are restricted by small scale parallel systems due to the overloaded scheduling, but the more resources we can allocate for our purposes, the more aggressively our defence system behaves. Thus, the probability of not observing a state transition within a round of observation is inversely proportional to the number of observers and the round intervals. It is also assisted by the "*probabilistic interleaving of operations*" [46] which is indirectly related to *time attacks* mentioned previously, by making the time slot for such a case smaller and unpredictable. As a result, this stochastic way of scheduling makes it almost impossible for an attacker to preconfigure non observable activities. In addition, the need to synchronize the observers and order the exchanged messages is underscored and taken into consideration. In this context, a number of events can cause the raise of an alarm indicating illicit activity starting from low probability conditions, up to missing infomation from the observation rounds and indication of violated observers. At that point it is vital to clarify that "*observational probabilites should not be confused with detection probabilities.*" [46], because the accumulation of the results works for us by posing detection much more probable with respect to the logical assumption that attackers desire the long-lived existance of any surreptious functionality they inject.

Although this mechanism is very powerful and describes the architectural bases for reliable sampling, the way this vast amount of intra-dependant information produced will be meaningfully interpreted is not directly answered. In addition, a more formal way of defining what is supposed to be legitimate states is also necessary. A more algebraic modeling of these factors is described by ***McEvoy and Wolthusen*** (**2010**) by providing a briefer way to express the expected behavior in a chaotic and complicated scenario as that. As a proof of realization, a modeling for a novel attack mentioned by ***Baliga et al.*** (**2007**) is modeled with the defined formulations. Special mentioning there is about the causal relations between the consecutive states. The importance of this point is paramount since it decisively determines the sampling pattern as it is recently proved by ***Seeger and Wolthusen*** (**2012**). Concretely, the dependecies between the attack steps and the inconsistencies they cause to our *semantic checkpoints* critically determine the way the memory accesses should be done and excludes a big number of concurrent observations by reducing the cost of observation as well. There is special consideration about the scenario where there is a data access racing between the observer and subverted host processing units, which is possible since the attack takes place during the inspection process. This might result in a number of unsuccessful metrics. These factors clearly assist in the false-positives minimization and raises the predictability of next volatile regions. At a next step, page locking of these loci can intercept attack completion.

However, this effective way of tackling the problem we face is not cost-free. We priviously mentioned the importance of the interval of each observation round. One could suggest these intervals to be as short as possible, to increase the probability of detection. As it will be explained, this cannot be done without respect to the cost caused to the normal operation of the inspected host. The reasons for this counterbalance and a model for cost measurement can be found in work done by ***Seeger and Wolthusen*** (**2010**). In a general notion, this aspect of the problem is very much related to the hardware architecture of multiprocessing systems. Mainly, for all the independent processors there is a memory hierarchy, which will be explained in more detail to the next chapters. Briefly, the model is structured from "top to bottom" starting from absolutely private and invisible regions to other units (i.e. the registers), resulting to a shared memory space accessible by every unit (i.e. the main memory). In the middle of them there is the cache memory which is uniquely assigned to each processor but it has a continuous interaction with the main memory. When a bunch of memory pages is loaded to a cache line and their content is modified, the hardware is responsible for the update of the deprecated values residing in the main memory. This is done by designated protocols designed for this task called *snooping algorithms*. This search for "dirty marked" memory pages and cause a *forced synchronization* [73] between the two memory regions resulting in the update of the related values. This synchronization is what causes a delay in the normal duty of the inspected system since the unit requesting for some memory block(s) is kept idle as long as the synchronization takes place. Cost modeling might become a bit more complicated if we think that the cache memory might me composed of multiple cache levels. However, we are not going to be concerned about that at the moment due to relatively limited amount of overhead added in this case.

Apart from the cost measuring formula, ***Seeger and Wolthusen*** (**2010**) provide some useful information about the contextual factors affecting these latencies, namely the amount of data under observation and their synthesis. The size effect becomes evident if we keep in mind that memory pages are not treated as autonomous units by the systems but they are loaded through memory blocks (or cache lines in the case of cache memory) and that the size of the memory blocks directly affects the sampling performance in a time unit. On the other side, understanding what the synthesis of information implies is not that obvious. In particular, the operations taking place (i.e. write or read) and the percentage of time spent on write operations decide if a memory page will be marked as *dirty* (read operations do not cause this) and if so what amount of time is needed for the synchronization of the values cumulatively. A proof of realization of this is documented by ***Seeger et al.*** (**2010**) and the statistic results produced are very interesting since they provide metrics for various configurations of the observations such as the size of data structures and the amount of them being altered. The conclusion that performance degradation is less if subparts of the observed regions are altered is very encouraging since a stealthy attacker would avoid to cause a generic subversion but instead a targeted one. We finally mention that answering if the intervals of observation should be variable or not is not easy. ***Petroni et al.*** (**2004**) and ***Molina and Arbaugh*** (**2002**) consider this fluctuation desirable by considering this unpredictability an extra obstacle for the attackers.

2.2.2 RQ2: How could we model kernel's flux and detect potential (non) control-flow violations?

Kernel manipulation consists of two attack vectors: the first is the redirection of kernel execution and the second one is kernel data subversion. The distinction between the two categories is not very clear since there is some overlap between them i.e. redirecting the execution of the kernel implies some kind of kernel data replacement (mainly function pointers). However, there are overthrowing attacks that do not change the control-flow of the kernel [10, 62]. RQ1 attempts to form a model of inspection on a superset of both groups of attacks but to do so, samples of legal values of the inspected kernel data elements are needed. In this research question we will mostly be concerned about kernel control-flow violations and the way the normal control flow can be modeled in a useful way during examination. Despite the fact that some attacks can hardly be detected, it is still very efficient since it still can localize a wide range of attacks because, as far as we know, most well known rootkits cause peristent kernel control-flow violations.

From the begining of our motivations we described the limitations posed by static analysis for various reasons with intensive interest on concurrent systems. However, there is a very active field of research working on this baseline for UNIX based operating systems. The reason is that the access to the source code of the kernel provides a number of advantages like types and variables exporting. ***Abadi et al.*** (**2005**) explain how Control Flow Integrity (CFI) can be enforced by injecting control checks before dynamic brances through a binary rewriting process. However, the authors mention clearly that their assumptions for **(1)** unique branch identification, **(2)** read-only code and **(3)** not-executable stack, become problematic "*in the presence of self-modifying code, runtime code generation, and the unanticipated dynamic loading of code*". Obviously, kernel falls into this category and additionally rewriting such a codebase is extrimely difficult due to its gargantuan control structure. However, it critical to think of the usefulness of a model injecting such flow integrity instructions both safe jump and return oriented.

Knowing these limitations of structuring a full kernel CFG beforehand, ***Petroni and Hicks*** (**2007**) presented a proof of concept implementation. Here, control-flow analysis can be devided in two parts. Firstly, the validation of the static parts, mostly by traditional methods like utilization of cryptographic mechanisms and secondly, the collection of the function pointers from the run-time memory regions of the kernel, the manipulation of whom can redirect the kernel execution to arbitrary locations. By validating the text segment of the kernel we can assure that the static branches (like unconditional jumps or function calls) are kept in tact. On the other side dynamic segment validation is a more complicated issue. Decision making upon dynamic kernel redirection, is dependant to run time state of the system such as heap, stack and registers. A virtualized environment is utilized for the monitor which has a number of assumptions that do not hold in coprocessing environments such as register accessing [53]. These information are used to exfiltrate data which can be traversed to access function pointers residing the kernel memory on the fly. To make such a desing generic enough, special consideration should be taken for both the way volitile items are accessed and the efficiency of the traversal algorithm as well. Finally, any address pointed by accessed function pointers must be examined and in case of a violation an alarm should be raised.

It is however clear, that transient violations might not be detected if they are not present during sampling. Some very sophisticated attacks of this kind can was developed by ***Wei et al.*** (**2008**) by taking advantage of dynamically scheduled interrupt requests through callback functions. In addition to the novel attacks the authors describe modeled countermeasures based on static analysis of the kernel source code. These papers clearly illustrate the power added to static analysis once the source code is available for this task and clearly reflect the extent up to which the kernel's design and implementation should be scrutinized to come up with the desired result. At this point, we should again emphasize on some generic restrictions placed by static analysis as discribed by ***Seeger*** (**2011**) steming from the modifiable design architecture of run time systems (in this case the use of $\mathrm{ptrace}()$).

2.2.3 RQ3: Could the stohastic building assumptions of a parallel sensor system become subtly fuzzy by a sophisticated attacker?

To this question there is no official or detailed answer documented. Thus, we will attemp to examine if an attacker with some knowledge of the monitoring strategy is able to take advantage of the probabilistic nature of kernel control-flow analysis, by hidding malicious transitions interposed with legal kernel jumps.

Chapter 3

Background Research

In this chapter we are going to describe a number of points which were not explained in detail in the literature review. We will start by explaining foundamental concepts of computer architecture with special focus on multiprocessing systems. An exhaustive explanation of them is beyond the scope of this project but it is vital to ellaborate on concepts that will be arising very often in our documentation. We will then describe some software issues of parallel programming before moving to a more precise clarification of the way Control and Data Flow Analysis works and what challenges are faced by these techniques when it comes to simultaneous programming.

3.1 Underlying Computer Architecture

Multiprocessing systems are very closely related to minimization of idling resources resulting in efficiency enhancement. However, it is critical to understand that the consequences from the parallelization not only provide higher utilization but also generate some security properties that axiomatically speaking vitally affect the effectiveness of statically analyzing concurrent programs. Generally, this aspect arises from the distinction between *concurrency* and *parallelization* as "*both are not interchangeable in the world of parallel programming. [...] In order to have parallelism, you must have concurrency exploiting multiple hardware resources*" [9]. **Parallel execution** implies a simultaneous progress for the software threads on separate resources while **concurrent execution** suggests a consequent interleaving between the software threads on the same unit. Bearing this distinction in mind we will focus our analysis in this section on the underground components of a concurrent system so that we gradually emerge these security properties. A generic observation is that parallelization gives distinct memory regions and formulates a run-time environment which is much more difficult to analyze both statically and dynamically. This environment is affected by the lack to precisely synchronize various processing units with a single clocking mechanism.

3.1.1 Multiprocessing Models

There is a big number of books documenting very comprehensively the models of multiprocessing platforms. Hence, we are going to focus on the basics for two reasons. We firstly need some primitive definitions that will be repeated throughout this documentation. Secondly, we need an intuitive understanding of the underlying architecture schemes since these formulate the platforms where we can apply the inspection models we are going to analyze.

In our case the main memory of the host system is the object under investigation because there we can find the kernel data structures that should be manipulated when the host is subverted. For this reason we fall into the category of **shared-memory multiprocessors** (SMPs). This category is devided into two subclasses mainly based on the organization of the memory and strategy employed to interconnect them. The first subclass is mainly known as *symmetric (or centralized) shared-memory multiprocessors* (SSMPs) where for all processors there is only a common centralized memory accessible equally by all units. This group is also known as *uniform memory access* (UMA) and implies that "*all processors have a uniform latency from memory, even if the memory is organized into multiple banks*" [33].

The second subclass is called *distributed shared-memory multiprocessors* (DSMPs) because the memory is distributed among the processors. This increases the number of supported processor counts because a system with centralized memory would not be able to provide the necessary bandwidth requirements without adding unaffordable latency to the system. These machines are also known as *non uniform memory access* (NUMA) multiprocessors because "*some memory accesses are much faster than others, depending on which processor asks for which word*" [56]. Apart from the increased performance and memory requirements by modern systems, the introduction of multicore processors[1], dictates the proliferation of NUMA architecture. However, the problem with this approach is intra-processor communication is much harder and as a result the programming design is much more complex. As it becomes obvious we will be referring to NUMA architectures.

3.1.2 Memory Hierarchy

The main challenge of NUMA architecures is the memory organization. As it was mentioned in the Chapter 2, the memory architecture of modern computers has a layered design which comprises of the next layers:

1. the **main memory** maintaining all the core components for the running system and the instructions to execute which is usually supplemented by the **swap space** on disk,
2. the **cache memory** the purpose of which is to store recently used memory pages by the processor in the cache, so that further requests for the same pages to the main memory can be avoided and
3. the **registers** which is the fastest memory region and fully isolated from layers 1 and 2.

The idiomatic behavior of this layered memory model in a concurrent environment is that not all memory components are transparent to every processing unit. All cores have access to the main memory, but layers 2 and 3 are invisible. In addition, cache memories comprise of multiple layers (most usually 3), say $L1$, $L2$, $L3$ and some of these levels are shared between the cores of a processor. As it is obvious, the page buffering taking place in the cache, in conjunction with the principle of spacial and temporal locality, critically affects the efficiency of the system for page accesses and nullifies the bandwidth consumption of the memory bus. However, this can lead to an issue known as *cache coherency problem* and as we will see it critically affects the efficiency of a memory patrolling system.

To understand the cache coherency problem we give a short description of the cache organization along with the related illustration at Figure 3.1.2 and an overview of the related

[1]which means that each processor consists of multiple caches

Table3.1: Notation overview for cache coherency

Symbol	Explanation
	Memory locations
p_i	memory page i
$b_{i,j}$	block j of page i
c_i	cache page i
$l_{i,j}$	cache line$j$ of page i
$[m]$	content of memory location m
$[m]'$	modified content of memory location m
	Transactions
P_i	processing unit i
$P_i \xrightarrow[t]{m} v$	processing unit i writes value v in memory location m at time t
$P_i \xleftarrow[t]{m} v$	processing unit i reads value v from memory location m at time t

notation in 3.1. Further details can be found in [36]. Main memory is organized into **memory pages**, each of them is composed of multiple blocks of memory, $p_i = b_{i,0}, b_{i,1}, ..., b_{i,n}$ where $i = 1, 2, ...m$ the related memory page. Respectively, cache memory is organized into **cache lines**, $l_{j,0}, l_{j,1}, ..., l_{j,v}$ where $j = 1, 2, ...w$ the related page. Here is a scenario where the cache coherency problem arises. Suppose that a memory block of page p_0 is cached by processor P_1 at time t_0 in cache line $l_{j,0}$ so that $[l_0 1] = [b_0 1]$. If at time $t_0 + \ell$ where $\ell > 0$, unit 1 changes the value of line $l_{0,1}$ so that $P_1 \xrightarrow[t_0+\ell]{l_{0,1}} [l_{0,1}]'$, we have two different instances for the same block, hence $[l_0 1] \neq [b_0 1]$. If another processor accesses the same memory page $P_2 \xleftarrow[t_0+\ell+\varepsilon]{p_0} [\cdot]$ where $\varepsilon > 0$, before P_1 flushes the updated content to RAM, a deprecated value will be retrieved. "*Notice that the coherence problem exists because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual caches, which are private to each processor core*" [33].

To tackle this problem, a number of *cache coherence protocols* are implemented at a hardware level. We will not dive into the internals of coherence protocols but we give a short description of the strategies employed so that we clarify why the inspection procedure generates big lattencies to the supervised host. Coherent caches can be implemented with two different techniques: **directory based** and **snooping**. Each one has different advantages and limitations with respect to the underlying chipset architecture. In snooping protocols the state of memory blocks is maintained in a memory location, while in snooping protocols the processor updating the value of a memory page is responsible for the broadcasting of the validity of the related page (set of the dirty bit to 1). Use of the bus as a medium to broadcast the invalidation or the updated value of a memory page is very convinient for both singlecore and multicore systems. The difference is that in multicore systems, the bus used is the one connecting the private caches ($L1$, $L2$) of the cores with the common cache ($L3$). As a result, when a write occurs, the processor performing the actions attempts to get access to the bus for the broadcast activity. The difference with directory based protocols is the maintenance of a bitmap table for each memory block where each bit indicates which units have loaded the related block to their caches. This makes broadcasting more scalable and targeted but in both cases, "*when an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the cache block invalidate it*" [33].

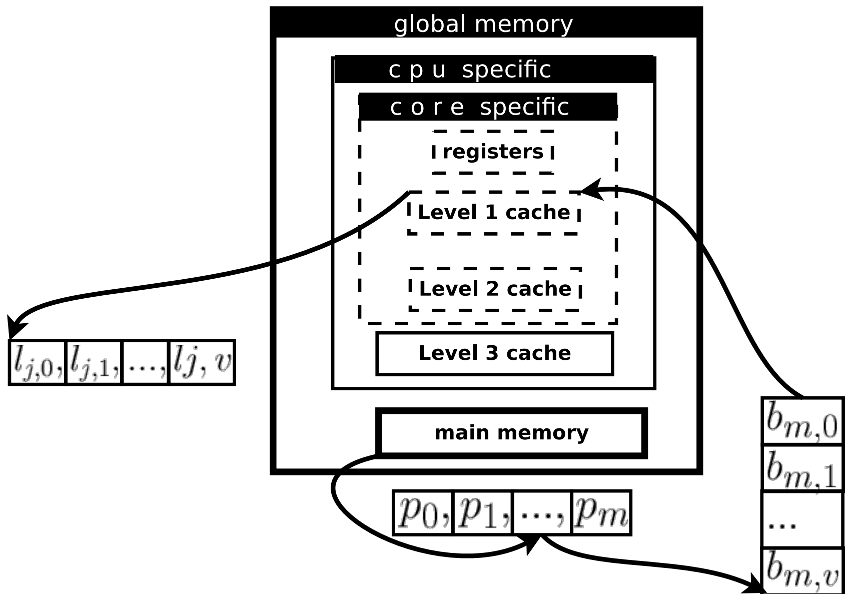


Figure3.1: Memory Hierarchy Model.

For an anomaly based IDS that supervises the state of certain memory pages in the RAM, the major consequence of the above mentioned solutions is obvious. For every round of data sampling by the sensors, there is a great performance degradation added for the host system. That is because the (co)processor willing to perform a memory access for inspection purposes (P_2), cannot proceed if the synchronization that makes the pages consistent is not finished. The more often the sampling takes place, the bigger the overhead becomes due to the increased number of *forced synchronizations* [73, 76]. In addition, as we will see in the next chapter, the size of the cache line along with the size of the observed data structure, critically influence the speed downgrading of the system.

3.1.3 Direct Memory Access

Devices decoupled from the main host offer a number of security and performance advantages and the procedure of memory accesses differs significantly as well. Data transfer methods between external devices and main memory divert. Initially polling and interrupts were the most dominant approaches. However, both are limited to low bandwidth operations because the CPU could be overloaded in cases of high workloading. For this reason, hardware designers introduced a novel offloading mechanism by utilizing a seperate device controller for data transfer "*directly to or from the memory without involving the CPU. [...] The DMA controller becomes the master and directs the reads or writes between itself and memory*" [56].

Furthermore, in DMA there are more than one buses involved in the I/0 transcactions (*input-output/read-write*). and their interconnection between them is architecture dependant. The first category is **synchronous** where a clock is included within the control lines and according to that a static communication protocol is responsible for the communiction [56]. Nonetheless, his type of memory access suffers from problems stemming from the inability to fully synchronize the clocks across multiple buses. For this reason, in DMA the memory accessing taking place are **asynchronous** because no clocking mechanism is involved. Instead, a sequence of steps is used to cordinate the bus transfers through a process also known as **handshaking protocol** [56].

Data transfers in DMA can happen either synchronous software pulling or by asynchronous hardware pushing. The two cases differ in the way the interrupt handling and buffer[2] allocation is managed.

[2]The buffer is the intermidiate bridge for data transfering and can be of types coherent and streaming.

DMA is mainly used when peripheral devices are connected to a host and the memory mapping is slightly intermingled. In essence, the hardware components based on DMA can use the physical memory of either the host or the bus and the mapping can be both linear (one-to-one) or intermittent. Data from the mapped memory is accumulated to the buffer and then handled by the external device. For more technicalities refer to work by Corbet et al. [28, Chapter 15]. As it also is evident, DMA introduces one or more paths to the main memory and this adds some extra parameters in the cache coherency problem which we will not examine here as it has to do with the possible buffer allocation strategies.

3.1.4 PCIe interfaces

An ubiquitous way to embed additional processing units within a running system is direct attachment of a Peripheral Component Interconnect Express (PCIe) card [20]. Such circuits are connected to the southbridge chipset[3] which is not directly connected to the CPU and the main memory. The data between the two peers is communicated through DMA by employing bus mastering protocols and the memory mapping and locating scheme provides full memory access. Further details will be given for the two schemas in the next chapter. Some extra characteristics are autarky in memory, storage and power supply which, as we will see, are critical parcels for an autonomous observer.

3.1.5 Graphic Processing Units

Another type of commodity coprocessor that can be attached to the main host are the Graphic Processing Units (GPUs). GPUs are the most proliferated, and the reason for this ubiquity is the great demand of the market for high processing for graphic purposes (mainly from the game industry). However, it is a very usual phenomenon to find GPUs used in every scope with intensive calculations and intrusion detection falls into this category.

GPUs are connected to the target system through the northbridge[4], i.e., the hardware component to which the CPU and the RAM are attached to. This gives an extra benefit to the already powerful processing capabilities, because it does not suffer from lattencies introduced in standard PCI cards due to their "distance" from the main memory and the lower bandwidth of the bus. Once again, data sharing is done through DMA bus mastering but for reasons of security the memory mapping is not complete, i.e., not all parts of the main memory are accessible. Consequently, although GPUs provide a very furtile environment for intrusion detection observations, there is a great number of constraints stemming from a number of architectural limitations [65]. We are going to list these limitations at subsequent chapters, before trying to come up with some conclusions about the conditions under which GPUs can be an effective anomaly based host IDS component.

3.2 Underlying Software Engineering

In a concurrent desing based on multiple processes (e.g. a forked server) there is a bunch of limitations. The first one is the difficulty to share data, since processes have distinct memory spaces. The second one, is the cost of the duplication process taking place with *fork()*. These limitations are surpassed my utilizing multiple threads. As it can be seen in figure 3.2 (based on the image found in [37]), the memory space of a process consisting of multiple threads is common for every thread. For the same reason, the process of thread generation is much faster as well. However, this benefit is not cost free. Special programming is needed to handle the dependencies stemming from

[3]Southbridge chip is named the I/O controller hub which is of lower bandwidth capabilities than the northbridge [56].

[4]Northbridge chip is named the memory controller hub and its main advantage is the high bandwidth of the provided interfaces (usually 4GB/sec) for the coupled buses.[56].

this thread data sharing. This is the most demanding aspects of parallel programming which actually deals with resolving the synchronization issues that might arise in a concurrently executed process. The source of this dynamic behavior is the non-deterministic CPU scheduling policy implemented by the kernel.

Concurrent execution might generate *data races* between two or more threads. Races are generated because of non-synchronized access to shared memory locations. Sections of code accessing shared resources for *atomic*[5] execution are known as *critical sections*. As we will see in the next section, static analysis of parallel programs is non-deterministic and the reason is closely related to the run-time behavior of synchronization techniques.

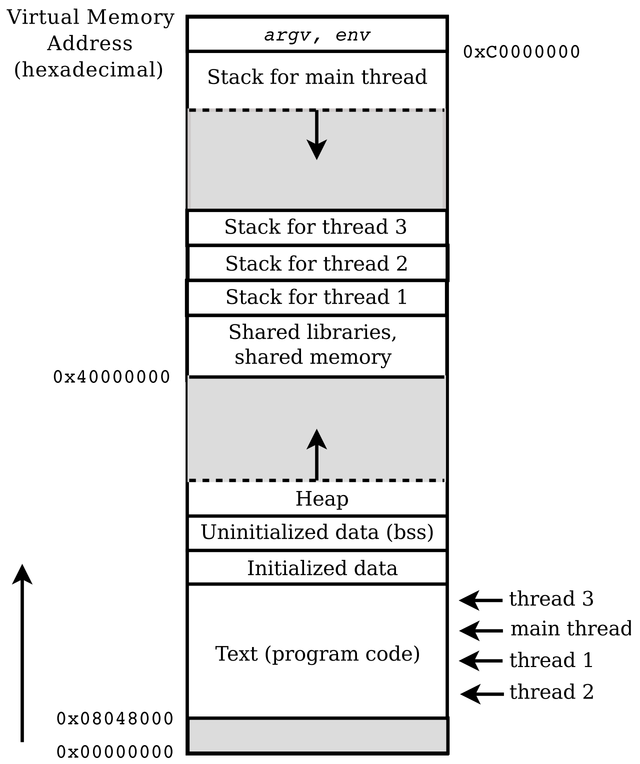


Figure3.2: Memory Layout of Multithreaded Process.

3.2.1 Synchronization

Synchronization is achieved by **locks**. Locks are implemented through mutexes. The mutex is locked while one thread accesses a shared resource controlled by this mutex. Locks require special consideration since bad handling might result in a **deadlock**. Even in the simple case of a single mutex a thread might result in deadlocking itself. This can be done if a thread attempts to lock $mutex_1$ twice. Things become more complicated if more than one mutexes are used. For instance, assume that we have $thread_A$ and $thread_B$ along with $mutex_1$ and $mutex_2$. If $mutex_1$ is hold by $thread_A$ and $mutex_2$ by $thread_B$ and $thread_A$ tries also to lock $mutex_2$ while $thread_B$ tries to lock $mutex_1$, then we have a mutual deadlock condition by the two threads. Deadlock are handled by carefully crafting the order of mutex locking.

In addition, locks can not only be achieved through mutexes but also using Reader-Writer locks. The main advantage of that is the fact that the available states for a thread are three (locked in read, locked in write and unlocked) instead of two (locked - unlocked). This feature gives some extra functional

[5]In terms of a software based approach, the term atomic refers to operations whose execution should be completed before any context switch

granularity that increases the degrees of possible parallelism. In essence, write-mode locks can be held only by one thread at a time while read-mode locks by multiple threads.

Finally, thread synchronization can be achieved using **condition variables**. In this case, instead of preventing access to a shared object by other threads, $thread_A$ can signal changes on a condition variable to other threads. In this way thread blocking (wait) is achieved, until a specific condition is fullfiled, after which the blocked thread can resume execution. Signaling in this case works as an indication of state change and, in conjunction with mutexes, condition variables help in minimizing the use of the CPU by blocked threads. For further details about the implementation aspects of thread synchronization in the *pthread* API, [37] and [80] are seminal resources for the UNIX Programming Environment.

Whichever technique a programmer employs for thread synchronization, there will be execution paths that will never exist at run-time. These full-duplex synchronization and communication conditions between two or more threads are known as rendezvous.

With these inference in mind, we move into a generic overview of static analysis. Once we finish with the introductory part, we are going to combine this conclusion with the characteristics of static analysis, so that the motivation described in the previous chapter, will be fully clarified.

3.3 Purview of Static Analysis

In the previous chapter we mentioned that static analysis is the main approach for the formulation of signatures characterizing a binary with malicious behavior. To make this more clear we are going to give a short description how this practically happens. Control flow graphs provide an abstraction of the source code and trees are the data structures used for the semantic representation of this flow [15, 16, 22]. As we will see each *node* in the tree represents a *basic block*, which actually is an autonomous code snippet consisting of consecutive instructions without any branches and conditional paths of execution. Any encompassed branches, either conditional or unconditional, are represented with *edges* and the initialization point of the execution is called *entry block* while the last one, *final block* [18].

3.3.1 Control Flow Analysis

At this point we can see an illustration of the above mentioned terms. An entry point of the W95/Bistro virus, as found in [81], is the following:

```
55 8B EC 8B 76 08 85 F6 74 3B
8B 7E 0C 09 FF 74 34 31 D2
```

Disassembling of the above sequence results to the assembly code snippet and the control-flow graph below:

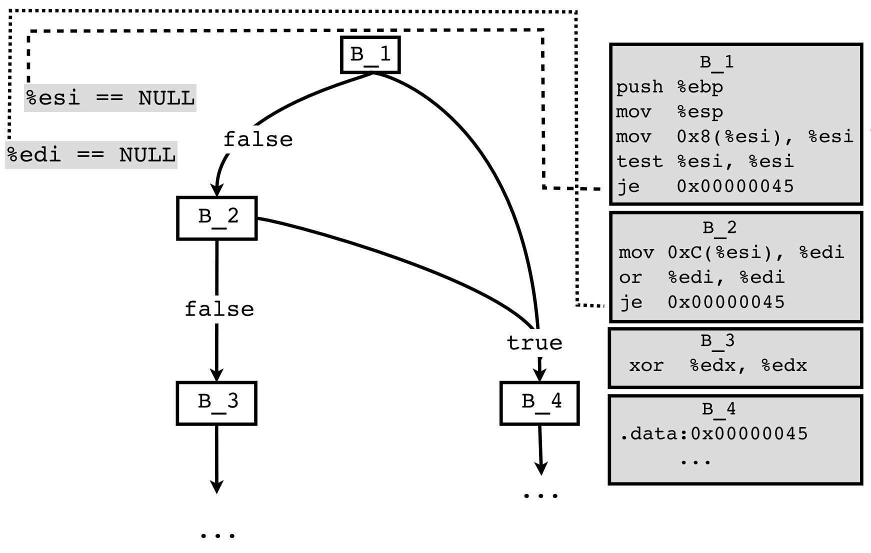
```
55
8B EC
8B 76 08
85 F6
74 3B
8B 7E 0C
09 FF
74 34
31 D2
```

(a) Hex byte sequence

```
push %ebp
mov  %esp,%ebp
mov  0x8(%esi),%esi
test %esi,%esi
je   0x00000045
mov  0xc(%esi),%edi
or   %edi,%edi
je   0x00000045
xor  %edx,%edx
```

(b) Disassembled snippet

Figure3.3: W95/Bistro original entry point signature



```
      B_1
push %ebp
mov  %esp
mov  0x8(%esi), %esi
test %esi, %esi
je   0x00000045

      B_2
mov 0xC(%esi), %edi
or  %edi, %edi
je  0x00000045

      B_3
 xor  %edx, %edx

      B_4
.data:0x00000045
     ...
```

Figure3.4: W95/bistro CFG.

Once a piece of malware is detected and analyzed dynamically, these information are obtained and its signature is generated, most usually by using a hash function upon an abstract representation on the CFG. As it is described by *Cesare and Xiang* (**2010**), decompiling algorithms can be used for the signature generation and several methods can be employed to optimize the signature searching algorithms. As we described earlier, the problem with signatures is that they are "brittle", which means that one byte change is enough to change the signature [22, 24, 52]. Such slide differences can be bypassed by using wildcards (similarly to regular expressions), but still more radical variations cannot be tackled.

As we said previously, various techniques are employed to scramble and obfuscate such signatures [81, chapter 49]. That could include packing of the executable [77, Chapter 18], adding identifiable redundancy without changing the behavior of the malware, code transpositioning etc [27]. Various techniques have been also implemented for the reverse process as well [19, 26, 86]. In the next image there is one variation of code snippet illustrated in Figure 3.3.

```
push %ebp
mov   %esp ,%ebp
mov    0x8(% e s i ) ,% e s i
t e s t  %e s i ,% e s i
j e      0x00000045
mov    0xc(% e s i ) ,% e d i
o r      %e d i ,% e d i
j e      0x00000045
x o r    %edx ,% edx
```

(a) Original Entry Point

```
jmp  s t a r t
jump4 :
   push  $0x58
   pop    %edx
   x o r    $0x58 ,% d l
   jmp  end
 s t a r t :
   push %ebp
   i n c    %ecx
   mov    %esp ,%ebp
   dec    %ecx
   jmp  jump2
jump3 :
   mov    0xc(% e s i ) ,% e d i
   t e s t  %e d i ,% e d i
   j e      0x00000045
   jmp  jump4
jump2 :
   mov    0x8(% e s i ) ,% e s i
   nop
   o r      %e s i ,% e s i
   j e      0x00000045
   jmp  jump3
end :
```

(b) Mutated Entry Point

Figure3.5: W95/Bistro excerpt

Figure 3.5 gives a brief illustration of the problems faced when dealing with obfuscated/transformed code. Apart from the problems added by the capability to compress or encrypt the payload of the malware, the fact that there are instructions with identical behavior and garbage directives makes the game of malware deobfuscation extremely hard for antivirus systems.

3.3.2 Synchronization-Sensitive Static Analysis

Given all the necessary background provided in the previous chapters we can now fall into the last aspect of this chapter. Understanding the limits of static analysis in programs designed to work in parallel is not a trivial task due the non intuitive behavior of parallel execution [48]. As we have already mentioned, the conditions that syncronize parallel tasks are known as rendezvous and each of these conditions represents a different path of execution that has to be belineated statically. The problem is whether a condition imprinted statically is bound to take place dynamically. The problem of creating a static analyzer capable of performing *synchronization-sensitive analysis* is known to be a problem $\mathcal{NP}$ complete by [31, 82]. In the same publication it is defined that: "*A static analysis is said to be synchronization-sensitive if it treads potential execution sequences that are not legal as unexecutable and computes information describing only execution along legal execution sequences*".

As a proof of realization look at Figure 3.6 which is a variation of an example illustrated at [82]. In this image we can see the existance of two threads. The main thread performs an loop that will be terminated at some point. Within the loop there are two if statements checking for the same condition variable. If the condition is not met the main thread is suspended, and resumes execution when the slave thread signals the condition variable change. When the first if statement (**I**) is completed, the same check will take place in the second statement (**IV**). In this case the second rendezvous (**B**) will never take place and as a result an execution path passing from point **V** will never be in place. That means that although the execution path **I, III, IV, V, VII** and **I, II, IV, V, VII** are depicted statically, it will practically never execute. This is because when the execution passes from path **III**, the main thread holds the lock for the mutex[6]. As a result the slave thread is blocked, which means that slaves_ter counter will not be changed. As a proof of realization a simple program in C can be found in section A.1.

Hence, it becomes obvious that things become more devastating for malware analysts when it comes to parallel execution of multiple threads across many cores. Even during dynamic analysis, heterogeneous instruction streams introduce nondeterministic behavior during execution. This makes debugging and reverse engineering of binaries more formidable. The reason is that debuggers transmute the scheduling behavior and this makes it possible conceal the authentic flux of the program. ***Ramalingam*** (**2000**) investigate the problem of performing a *context-sensitive synchronization-sensitive intraprocedural static analysis*, that is proved to be undecidable. In the same parer it is defined that: "*An intraprocedural analysis is said to be context sensitive if it considers only paths with no mismatching call and return edges*".

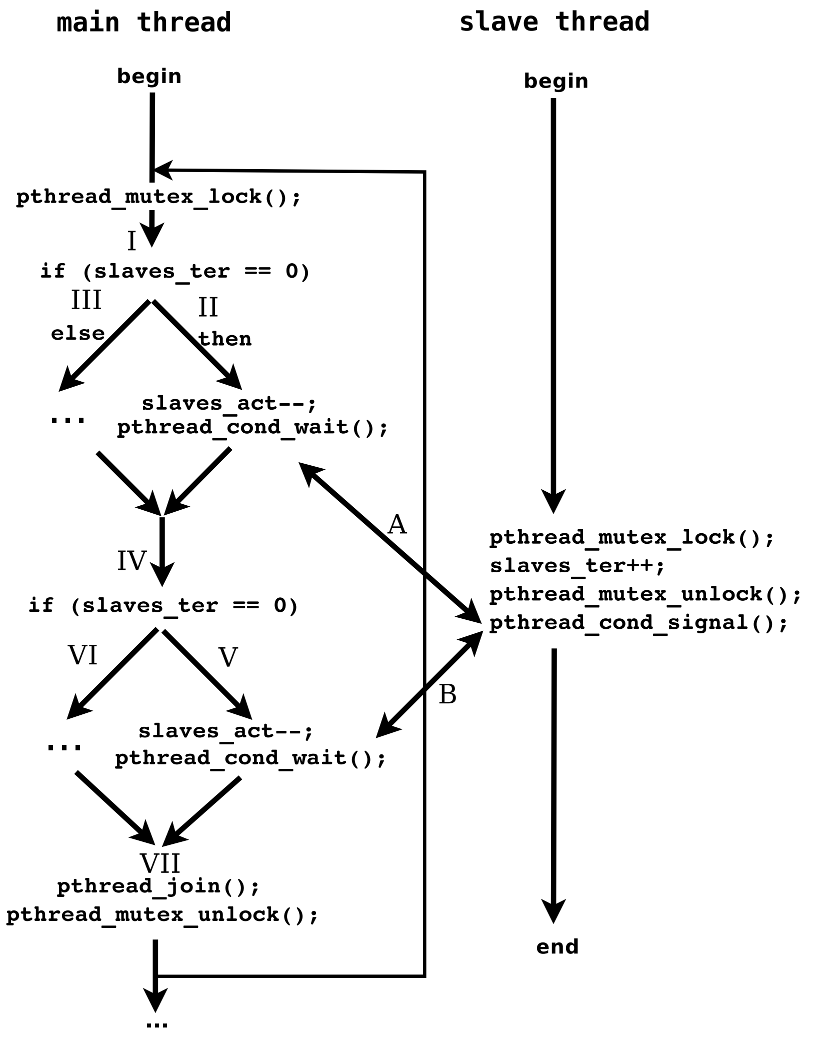


Figure3.6: Static analysis of parallel execution.

Finally, one of our assumptions is the ability to distribute the executed threads to different cores for reasons related to the security properties provided my parallel systems (see section 3.1). The importance of thread distribution is important for one more reason. To improve performance, the operating system (and secondly the hardware) will put efford to keep threads of a multithreaded process in a single unit. In this way procedures like cache line updates when a thread is moved from

[6]the mutex is unloced by the **pthread_cond_wait() function** , it is then locked back, when the condition is met (i.e. on **pthread_cond_signal()** call)

processor A to B are avoided. This implies that systems fighting malware (e.g. antivirus), most probably will not occupy all processors/cores of a running system and as a result some memory regions will be (almost) inaccessible. We use the word almost because although registers are physically isolated [53], the private cache of a core (say $L1$) can be tuned not to evict certain memory pages for optimization. That lets a malicious process/thread to run an algorithm without revealing the internals of the implementation to an inspector of the main memory. Of course, nothing stops an observer from polling the related memory pages from RAM and cause a synchronization to take place. However, if the memory location is randomly chosen by the malware, using some in-register stored parameter for example, then the defender cannot decide which memory location to examine. The only alternative to this obscurity scheme is full memory dumping, which is rather painful in terms of performance degradation. The *pthread* API provides the necessary interfaces for this (look at **Listing A.1** – related illustration in 3.7, based on related image found in [9]). In multiprocessing models, there are also a number of issues that might have to be solved like data transfering strategies, notifications and parallel processing. The latter can possibly be the more complex that the previous because it requires predictions for equal workload partitioning. The basic prototypes are the Master/Slave and Data Flow models ([6] is a good introductory point for an inclusive outline).

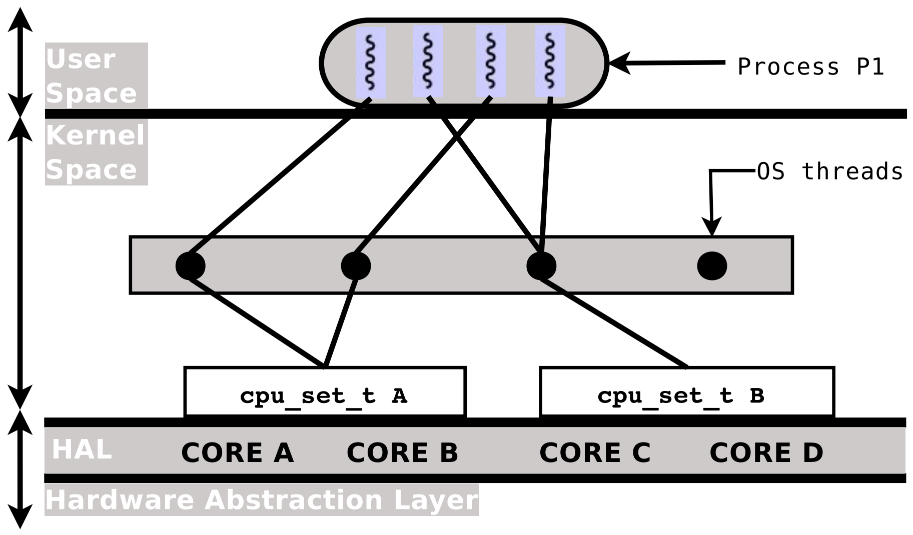


Figure3.7: User space thread mapping scheme.

Conclusively, it becomes clear that a singature-based approach for detection of concurrent malware specially crafted to take advantage of the limitations of static analysis, as described in the sections above, is obviously not the most effective approach. Consequently, alternative strategies should be employed. This includes either behavioural or anomaly-based Intrusion Detection Systems. Behavioural analysis looks for sequences of events (e.g. system calls for connection establishment, file opening, process execution e.t.c.) and make decisions upon the results found. On the other hand, an anomaly-based IDS looks for sequences of changes upon some observed data, and variations from the legitimately modeled layout, is marked as suspicious. In the next chapter we are going to examine:

1. what the prerequisites and the challenges for such an IDS are,
2. which inspection models are available and
3. an overview of each model, mentioning the related pros and cons.

Chapter 4

Concurrent Memory Inspection

There is nothing more difficult to plan, more doubtful of success, more dangerous to manage, than the creation of a new system.

Niccolo Machiavelli

A foundamental decision that must be taken upon an anomaly detection systems is the determination of the necessary model that should be finally utilized for inspection purposes. The model implies where the host IDS mechanism is located and consequently in which ways it performs the data extraction and communication with the observed machine. Before giving an overview of the available models, we enumerate the properties that such an IDS should fulfill. Some of the properties below overlap with some of the requirements listed in [65].

1. **Full Memory Access**: Since the core elements under investigation are the kernel data structures, full access to the host memory is necessary. Access to kernel space addresses might be restricted.

2. **Reliable Memory Extraction**: The content of the captured memory pages shall never be invalidated either deliberately or unintentionally. The reliability of the memory snapshots are the most critical factor for the validity of the results.

3. **Resilience to Subversion**: The IDS system could possibly be attacked directly. To reduce this possibility we can either physically isolate the sensors or populate them accors the system.

4. **Processing Power**: There is no doubt that the observation process is expensive by means of processing time. The degradation is also relative to the **host workload** since a congested host performs memory accesses intensively. To reduce the impact of this to the supervised host, the sensor(s) should be of high processing capabilities and possibly flexible enough to the sampling density.

5. **Permanent Activity**: The memory inspector must be able to work in parallel to the underlying machine during its whole lifecycle. There must be no intervals during which there is no supervision system active. This is critical even for the bootstrapping process, where the initial trusted state of the system is established.

4.1 Sensoring Models

The above mentioned primitive factors severely affect the decision upon the underlying model. In this section we present the possible models that could be utilized along with a related advantages and disadvantages of each one. The conclusions made upon each prototype are connected to the conditions listed at the begining of this chapter.

4.1.1 External Monitors

As it might have already been observed, the research for anomaly detectors started with external monitors [49, 61, 87]. This option could be considered both secure and convinient, despite its limitations. The first aspect, which is very valuable for safety reasons, is the fact that the security of the mechanism does not depend on the security of the observed machine. The physical seperation between the sensor and the host gives absolute independence to the detector. Concretely, the effect of OS vulnerabilities found in the host is limited and most probably absolutely harmless because of their seperate resources (i.e. processing units, BIOS, firmware and memory) [74]. Of course, this might not be always valid and this depends on the type of the sensor, as we will see subsequently.

The convinience, from the other hand, implies that systems that could be used for external monitoring, are omnipresent and at some cases pre-installed in the PCs (e.g. GPUs). There are multiple types of widely used coprocessors like PCIe, GPUs, Filed Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs) etc. We are going to present the first two of them, for peripheral scanning and we will present the idioms of each case seperately.

4.1.1.1 PCIe add-in cards

In section 3.1.4 we gave a brief description of PCI primitives. Peripheral Component Interconnect Express (PCIe) cards has been utilized for host intrusion detection purposes in various works [12, 23, 61]. Such a coprocessor includes various valuable characteristics like powerful processing, autonomous main memory & storage and non-volatile power supply. The memory access takes place through DMA using bus mastering protocols and the decision of the memory regions that are about to be accessed, are facilitated through data communication between the two parties. To keep things simple the set-up for the memory mapping is a simple one-to-one scheme which means that a PCI memory access corresponds to the relative 32-bit address of the host. This results in full memory access and no reverse mapping by the OS is necessary.

The only problem that has to be fixed is the computation of the host's physical address from its relative logical (virtual) address by the coprocessor. The reason that makes this a problem is the zero knowledge of the PCI about the virtual addresses translation scheme of the host system.[7] To do that the PCI performs an emulated translation of the desired virtal page, and then accesses it directly through the bridge. As it can be deduced by **Figure 4.1**, for this translation to be correct, the right page table must be accessed in the host's memory. In this way the memory snapshots are located and then copied through a sequence of steps to the userspace of the observer.

As we can see properties **1, 3, 4** and **5** are fulfilled and afterwards we will present why property **2** is questionable.

[7]The mapping is done upon the physical memory of the host and for that only the physical memory is visible by the coprocessor.

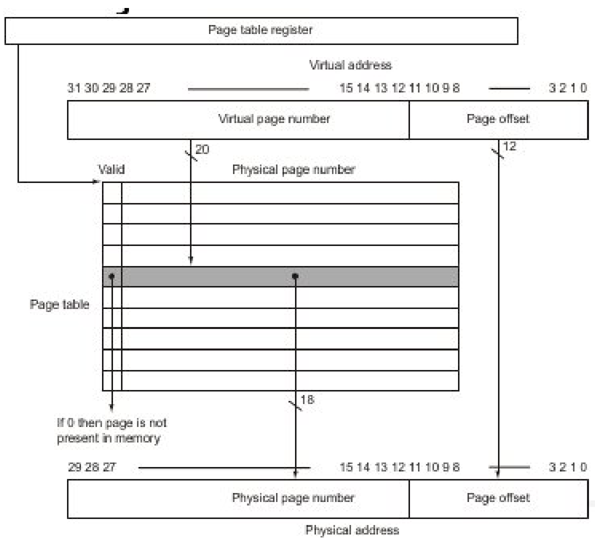


Figure4.1: Virtual Address Translation Mechanism as illustrated in [56].

4.1.1.2 GPUs

GPUs' basics were provided in section 3.1.5 and the first point distinguishing it from standard PCI coprocessors, is the ultra-forceful processing capabilities and the fact that it is internally embedded to almost every modern system. As we expained previously, apart from the hyperthreading architecture, the locating of the GPU and bus speed assist in the performance accelaration. However, GPUs suffer from a number of limitations [65] starting from the lack of full access to the host memory due to the restricted and proprietary APIs provided by the major industry vendors (i.e. NVIDIA & AMD). To overcome this problem, the only possible solution (for now and the foreseable future) is use of native GPU assebly interfaces.

Furthermore, GPUs do not consists of non-volatile storage, useful for incident logging purposes and also they consists of no non-volatile power supply. The latter is a matter of high importance because the **5**th property is not met and as a result we are facing the problem of bootstrapping trust [55]. Attacks that could be based on this constraint could be BIOS or kernels' flashing, i.e., it is possible for the CPU, which initiates the communication channel between target and observer, to freeze inspection kernels or reload the Basic Input-Output System (BIOS). A solution to the former issue, is manual change of the GPU firmware[8] so that kernels which are actually present by means of hardware but deactivated by the vendor, can be utilized. As for the latter, although such an attack – to the extend of our knowledge – is not yet presented, there is no good way (bullet proof security formula) to eliminate it. A proof of concept implementation covering and discussing all the above mentioned facts is done by Seeger et. al. [74]. In general, as we have seen although GPUs are very powerful coprossors that use a PCIe bus for interconnection, they do suffer from various compulsions not in place for add-in cards and only property **4** is a built in characteristic that is 100% fulfilled.

4.1.1.3 Shortcomings

In the previous paragraphs, we did not explain why property **2** is doubtful for out-host auditors and this paragraph is mainly concerned about this aspect. Taking a memory snapshots through a DMA mechanism from a hardware-based acquisition mechanism has a number of advantages like:

- direct communication with the memory controler,

[8]Such a process apart from complicated also lacks of detailed documentation.

- it is CPU independent, which means that it keeps the system degradation at minimum and
- compromisation of the OS does not affect the ability to get the snapshot.

However, the isolation stemming from the decoupling of the observer from the target might cause severe limitations and the grade up to which the resulting image can be absolutely reliable is under question. At first place the way the address resolution takes place once a request is made, can result in acquiring falsified memory images. The first aspect has to do with the poisoning of the registers used for cached virtual addresses in the Translation Lookaside Buffer (TLB), to limit memory accesses in the page table [56]. This approach, althought efficient, gives to an attacker the ability to manipulate the TLB entries and return falsified memory addresses in case of a read request (instead of execute). This is possible in Intel Architectures where two seperate TLBs are maintained, ITLB (Instruction TLB) and DTLB (Data TLB). This can be fatal for the snapshot validity because an attacker can subvert only some entries, by redirecting memory accesses to pages revealing malicious functionality. An implementaion of such an attack can be found in the shadow-walker rootkit [78].

The second aspect is mostly related to the fact that DMA accesses are handled by different units than those by CPUs. This distinction can result in redirecting forged memory contents. This requires reprogramming of the chipset responsible for the address translation where in essence an intruder is able to remap physical memory address space to the bus space. A proof of concept implementation for AMD architecture is shown by Rutkowska [68]. **Generally, caching memory transalations to reduce access penalties or seperation of memory controller between the CPU chipset and potential coprocessors can create stealthy ways to incapacitate memory snapshots.** In DMA mechanisms the signals exchanged for data transmition, travel across multiple controllers and bridges, which increases the possible locations for man-in-the-middle attacks.

In addition, the behavioral design of DMA can result in inconsistencies generated at run-time. The fact that the main memory continues its normal operation and that the monitor cannot lock the accessed pages since they are accessed through DMA, means that while the snapshot is taken, the memory contents do not stay constant. As a result there is a great interest for the speed of memory copy, despite the fact that the regions are copied once they are locked. Finally, it is vital to clarify that the risk of DMA acquisition is the changing of the scheduling time because the CPU is haulted from the use of the bus.[9] This is done by special signals available, that actually make the PCI to be the bus master, instead of the CPU, for a time slot. These reasons make the existance of a race window obvious, since **page locking** guarantees that the memory pages in question will not be swapped to disk but overwritting is still possible. Kernel wide-lock mechanisms could be used to hault any other activity but it becomes obvious that this introduces an unaffordable overhead and neglects the original mission of DMA.

We end up our observations by reminding that in the memory hierarchy model, the registers of a CPU are not accessible by other (co)processors. If we recall from **Figure 4.1** the page table registers contain the physical addresses in which the page table responsible for logical to physical memory transalations can be found. Especially in the case of the PCI add-in card, this is necessary for the translation process described in section **4.1.1.1**. To solve this deadlock, PCIe uses well know addresses of the main memory where the page table can be found. Obviously, and change to these register would cause a big problem to the memory translation process and consequently to any out-built based sampling mechanism.

Digression

At this point, that kernel control flow analysis is not explained, it migth not be very clear that the memory images obtained by an observer do not provide full observability of the host. Dynamic

[9]As a consequence of the bus mastering protocol.

kernel elements do not have a deterministic behavior and that requires not just matching their values with some known predicates. Instead interpretation of the results is necessary for obtaining a detailed view of the system. In addition, it is implied that the memory snapshots might not be precise not just because of malicious activities but also due to run time inconsistencies. This probabilistic aspect of the analysis generates some extra gaps that will be mentioned in the next chapter.

4.1.2 Internal Monitors

Establishing a detection mechanism within the host under investigation, provides a number of advantages for both security and efficiency reasons [45, 46]. From a security perspective in-host monitoring has full access to the main memory and it is not vulnerable to attacks that could invalidate the contents of the memory snapshots. Attacks like the one implemented by ***Rutkowska*** (**2007**) are not possible because the buses interconnecting multiple processors, as far as we know are not re-programmable. TLB poisoning however, might still be possible (despite the limitations mentioned by ***Sparks and Butler*** (**2005**)) but it is much more complex. Generally, since no DMA is needed for the snapshots, the page fetching is more precise and resistant to manipulation because signals go through a more limited set of hardware components. In addition, the memory copying is faster and the possibilities for inconsistencies decreased.

In terms of efficiency, although normal processors are not as powerful as coprocessors (like GPUs for example) and the host system is responsible for more tasks, the tradeoff might be counterbalanced if the number of available processors/cores is big enough. Furthermore, the fact that the mechanism has the ability to interpose itself with the operating system, gives it a clear view of the activities running in the machine and it is easier to trace malicious activities and analyze them in more depth. Finally, although obvious, it is noteworthy that an internalized monitoring tool, works contiguously directly from the boot-up up to the termination process.

4.1.2.1 Shortcomings

These advantages come at a price. A non-physically, through isolation, protected mechanism like this, is fully exposed to subversion itself. This includes either exploitation of the sensor or man-in-the-middle attacks on the message exchange framework. The self protection of the mechanism shown by ***McEvoy and Wolthusen*** (**2010**) is based on a number of observers examining the data structures of the monitor itself. As it is shown, the bigger the number of observers, the higher the probabilidy to detect instant changes on the observed data. In Linux systems, this implementation is possible through Linux Kernel Module (LKMs). Each observer exchanges and compares the acquired states with its peers and after comparison, any inconsistency is logged. To subvert the communication infrastructure, an attacker faces "*a combinatorial barrier to subversion*" due to the multiplicity of the available observers. That means that $n \cdot (n-1)$ messages must be manipulated to hide the subversion of the exchanged messages within a round of observation, where n the number of observers. This along with the *nondeterminist concurrency*[10] of the sampling process make it extremely difficult for an attaker to spoof exchanged messages in a consistent manner.

However, we should recall that LKMs export symbols that can be touched by both other modules or patched kernels. Since kernel modules use hooks, any symbol exported from the module itself for "private or public" use, can legally be used by another malicious module (e.g. rootkit). This makes it possible for an intruder to hook the observers directly or insert a filter driver that purifies the messages sent among the ensembles. The difficulty of such a straightforward attack is the assumption that the systems starts from a trusted state. This is because the observers are activated from the boot-up process and as a result, to the best of our knowledge, an attack subverting the systems during initialization is not possible.

[10]With the term non-deterministic concurrency we imply the physical inability to predict the excecution order of threads an as a result the assortment with which messages will be exchanged.

Also, use of dynamic loadable modules is discouraged (e.g. Kprobes) and if so it must be verified to be signed, to perform Mandatory Access Controls (MAC) at load time, e.g., by using EVM/IMA functionality [2] and finally as it is already mentioned, integrity checks after load time. In addition, it should be of considerable notice that, although in-host, the mechanism based on kernel modules could also cause race conditions with the accessed data. Concretely, accessing data from within the system while it is modified, it might give inconsistent/garbage data or even cause kernel hang (segmentation fault). This happens because the kernel structures are handled by internal kernel mechanisms that take responsibility of various issues like synchronization. Threads that are not handled by the synchronization mechanism responsible for the management of the related kernel data, can cause access violation conditions. To overcome such problems, very careful handling of the shared data accessed must be implemented. Extra attention must be paid if accessed data structures are managed by functions handled by a state machine with preemption enabled in the kernel [17]. A last scenario that can also cause serious misbehaviors is when a processor tries to access a memory region which is currently DMA memory mapped for an external device for example. In this case, as fas as we know, the available interfaces do not provide a way to perform checks upon the mapping status of a memory locus.[11] For this reason, synchronization can be only handled by special types of locks, called *spin locks* [44]. Status checking upon these locks can only be done through interaction with the driver. A mapped memory section cannot be accessed by the driver while the device is pushing data to it because it belongs to the device [28, p. 449].

Last but not least, for the mechanism to be resilient to attacks, a big number of processes and threads for self inspection is needed. Evidently, such a technique consumes a lot of resources that might not be affordable in small scale systems and when implemented it requires load balaning. Another consequence of this is the existance of long intervals where many resources remain idle. Furthermore, for reasons that will be explained later on, the impact of the performance degradation will also be increased proportionally to the number of observers. Hence, such a model is designed to serve and can be reasonably applied to large scale systems.

4.1.3 Virtualized Monitors

The final option we are going to mention for memory inspection purposes is a virtualized monitoring system. Virtualized environments provide an emulation of a host system without use of any extra hardware. Virtualization is of increased popularity because of the raised need for isolation and resource sharing in modernized systems. The hardware-level abstraction provided can then be utilized and managed by a **Virtual Machine** (VM) and VMs are supported by special software called **Virtual Machine Monitor** (VMM) [56]. Due to space limitations we will not go into further depth but a lot of resources are available for the interested reader [5, 66, 67].

The main advantages of such a sensor is its ability to have full memory access to the target system, even in the registers of the CPU(s). The latter can be done through a *hypercall* invocation which can use a passed buffer for this mission. The memory mapping is follows a similar flow as the one described in paragraph 4.1.1.1 but register access by the hypervisor makes the whole process more reliable. To the best of our knowledge, there are not any known attacks that can invalidate the contents of a memory snapshot taken by a VM. In addition, the sandbox within which the monitor runs keeps it isolated from the underlying systems and as a result it will not be affected by host subversion. ***Loscocco et al.*** (**2007**) and ***Petroni and Hicks*** (**2007**) provide implementations based on Xen hypervisor for kernel control flow manipulation detection and we will further examine this tactic in the next chapter. Also, ***Srivastava and Giffin*** (**2011**) employ the same method for detection of benign kernel drivers by arbitrary calls triggered by malicious drivers.

[11]This is mainly because it is not easy in all architectures to check the mapping without accessing the memory.

4.1.3.1 Shortcomings

The weak points we could mention about such a system is the great performance impact added to the system. The context switch needed in every cycle of observation reduces significantly the throughput of the system in question. To reduce the cost of virtualization, programmers should follow various guidelines [30]. We stress out that the performance impact is much bigger than in the previous models. For example, ***Loscocco et al.*** (**2007**) show that for a measurement frequency of 2 minutes –rather long interval compared to the previous cases – the host performance was reduced by approximately 15%. A similar conclusion is deduced by the results presented by ***Petroni and Hicks*** (**2007**). Obviously, the performance impact is closely coupled to the limitation of performing rather "infrequent" sampling and we described later on (paragraph 4.2.2)the reasons for which we want to avoid such a case.

Moreover, hypervisors can suffer from software vulnerabilities [12] most of which result in DoS attacks, with obvious consequences for an observer. Furthermore, there have been documented attacks that add a malicious VM or hijack an existent one, either by relying on commercial virtualization (SubVirt rootkit) [39, 85] or hardware virtualization technology (Bluepill rootkit) [69]. The latter is a way to perform memory remapping similarly to what we described in 4.1.1.3 but for Intel Q35 chipsets. In this architecture, the processor allows manipulation of the REMAPLIMIT and REMAPBASE registers, which are normally set during start-up by the BIOS, and can lead in a malicous Dom0 kernel accessing the memory of the hypervisor. Although there have been proposed countermeasures against such attacks, e.g. exclusive lock of remapping registers by the BIOS [58], we see that there is always a possibility to have novel attacks for virtualized monitors also.

Under the assumption that, a virtualized monitor is not capable of running from the very beginning of the boot-up process which leads as to the same problem described in paragraph 4.1.1.2. As for the attacks we descibed previously that spoof the results of memory snapshots, although there are not related attacks for emulated environments, there is no good way to prove that it cannot happen. The reason is the existance of intermediate levels responsible for memory mapping that can possibly be manipulated.[13]

4.2 Observation Cost

In section 3.1.2 we explained qualitatively, how the solution to the cache coherency problem introduces delays through a memory monitoring system. In this section we will give some further details through a more quantitative approach. Practically, the degradation is not fixed and a number of factors affect the extend of the performance decrease. These factors are:

- **Memory Access Patterns**: In an asynchronous monitoring system the sensoring model affects the patterns utilized for memory accessing. With respect to the underlying protocol utilized, the preperation set-up and the synchronization procedure add some extra latencies to the access time.
- **Cache Organization**: This includes all the parameters that decide the behaviour of the cache memory. The first is the **size of the cache line** which is also connected to the number of the cache lines per page. The second aspect is the algorithms and the data structures formulating the **assemblying method of both the cache lines and working sets**.
- **Data Composition**: This encompasses the **sampled data size and mode**. Concretely, the

[12]E.g. various CVEs have been assigned for the two widely used open source tools – Xen & KVM.
[13]This is similar to implementing TLB poisoning

size of the observed data structures and the size of the cache line, determine the number of units to which the sampled data will be fitted into. With the term mode we mean whether data is ***writable*** or ***readable***.

- **Observation Frequency**: Once again, the rythm with which the observers poll samples from the inspected memory regions, is analogous to the forced synchronizations triggered by this event. The more we measure the less efficient the system becomes but the more precise the results can be.

With respect to all these dimensions we analyze the variations of the cost impact on a supervised host in terms of performance downgrading.

4.2.1 Cost Meassurement

The analysis of cost measurement would be very easy if we only had to care about how often the observations take place. More than that, we have to look ahead as the extend of these actions is determined by the combination of the previously listed items.

Keeping in mind that the synchronizations enforced by the cache coherence protocols, are done on a cache line sized granularity, the minimum bandwidth consumption on the memory bus per synchronization is decided by the size of the cache line. The message this behaviour implies is clear: the bigger the line size, the higher the bus contention becomes and it can possibly have an impact on the duration of the block exchange. For example, think of a line of 64 bytes consisting of a checkpoint under inspection. If a change exists in the first 16 bytes then the bus is loaded with 48 additional bytes. Alternatively if the cache line was 32 bytes long the overhead would have been 16 bytes (100% overhead instead of 300%). The impact on the synchronization speed cannot be precisely predicted because it depends on the type of the hardware and its architecture but it cannot be marked as negligent.

Moreover, when we have a large cache line it is more probable to have unrelated data residing in the same block. Accordingly, there will possibly be block transfers triggered for coherency reasons, that will be caused by modifications on blocks of no interest. As a variant of the previous example, think that our checkpoint is a 16 bytes long variable and the 48 extra bytes are heavily modified data, irrelevant to our observations. Every time the sampling process exists, there are synchronizations that should not normally be in place. This problem is known as **false sharing** [33, 38, 56].

Therefore, the cache line organization granularity on top of the data size, play a significant role in the overhead added to the system. For the same reasons, manufactures tend to reduce the cache line size for optimization reasons and there are various techniques at both compiler [21, 29] and software developement level [3, 4, 41, 71] (padding, ordering aligning, packing etc.), which assist in reducing the possibilities for such conditions.

Finally, one more point to explain is that only writable data can cause synchronizations because read only data are not subject to changes. Actually, for every time span $[t]$ only the percentage of writable data that was finally altered causes synchronization runs. A formula for the *interference ratio* based on this factor can be found in [73] along with a proof of concept and the related statistic analysis in [76]. For this reason it can be also considered as a good practice to group all the read-only data in seperate cache lines.

The above presented points cover the aspects of top bullets 2 and 3. The explanation for the 1st point is architecture specific. Generally speaking, off-host observers not only cause cache synchronization but also face some extra overhead because of the additional steps involved due to DMA. Apart from the memory mapping taking place at start-up and the extra data exchanged during memory accesses for communication purposes, data transfers are affected by three crucial metrics, i.e., *latency*, *resource contention* and *bandwidth*.

In our case, the bandwidth of the bus is not the real source of the overhead, since the sampled data will hardly throttle its capacity due to their limited size. However, resource contention is somehow connected to this because whenever the observer goes through some shared resource (e.g. a bus or a cache), objects other that our observers are restrained partially from using the component. On the contrary, latency is affected by numerous factors[14] and it should be seriously considered since when small data chunks are transfered, the impact is higher. That is because "*the latency is a constant startup cost, which must be paid regardless of how many items are to be transferred*" [35]. Since we transfer small data items during our observations it comes under no suprise that a repeatitive cost is bound for systems of our interest. Moreover, modern memories are signle-ported dynamic RAMs (DRAM) and as a result we cannot perform multiple read/write operatations simultaneously (we cannot benefit from deserialization of actions).

Furthermore, we should consider what the minimux data transfer size is. If the external observer is a GPU for example, then the standard APIs provide quite large minimum scaling (8K) [34], presumably because CUDA for example use a non pagable buffer of the related size. On the contrary PCI add-in cards are more granular and the contention of the medium will be less for each transaction. Finally, it is worth noting that especially in modern GPU architectures, the access time for the global memory is not always analogous to the size of the data block size, but on the contrary some times it is almost fixed [1]. Although the diagrams provided in the source cited before do not include the time needed for data transfering from the host to the GPU, we can reasonably infer that overall latencies are bound follow a similar pattern.

4.2.2 Frequency of Observations

The last aspect for this section is coming up with some conclusions about the intervals between the various observations. It is clear that reducing the intensity of the sampling procedure reduces the performance impact, but it is also evident that long intervals between the data pollings can assist in having clandestine activity that conceals its presence by restoring periodically the manipulated data. In addition, the frequency of observation rounds is a case dependant decision. Namely, all the afore mentioned points (e.g. cache line organization, the size of the observed data etc.) as well as the interconnecting model between target and observer, should very carefully be analyzed before a decision making. Especially in the case of a coprocessor based model, the memory copy actions are more time consuming with respect to the transmition capacity over the DMA channel. "*Transferring small chunks of data is faster that transferring bigger ones*" [76].

For this reason, in any observer this interval should be configurable and possibly not fixed at all. This implies that both for reasons of *impredictability and adaptiveness*, the examination regularity could be randomized (fluctuating between a predefined time slots) [49, 61]. By impredictability we mean the physical inability of an attacker to perform time based data reseting due to the lack of predefined intervals, while by adaptiveness we imply the fact that the *semantic checkpoints* [47] can be updated during observations. Although the former is not difficult to understand, the latter is less intuitive and we will most thoroughly analyze it in the next chapter. Briefly, for a sophisticated attack to succeed, various actions of kernel data manipulation must take place. These manipulations are not arbitrary but a causal relation between them exists [45, 46, 47, 75]. A memory "patrolling" mechanism aware of these causalities, can perform less heavy memory inspection and direct its observations against subsets of the checkpoints. Minimizing instantly the workload of the observer significantly "***reduces the performance degradation on the host-side due to less interference***" [75]. In the event of a subversion, the subset under surveillance changes and so does the processing time during the inspection rounds. For this reason it might be desired to either expand or shorten the frequency of observations.

[14]Such actions are instruction decoding – by the CPU because normally the CPU should decode the instruction and then signal the GPU to perform the copy – waiting for becoming the bus master and latencies caused by the memory.

Addendum

The information elaborated previously constiture a **snapshot of the current state of art**. Each previously presented model offers different advantages and disadvantages which can change due to the rapid advance of modern architectures and this dictates the need to **stay up to date** with the related technologic developments. Table 4.1 gives a qualitative digest and as such, it could be arguable that the characteristics deciding the thresholds are under question. However, *mutatis mutandis*, it gives a comprehensive representation of the prons and cons for each model on a compare and contrast basis. For this reason, there is no good way to decide which is more suitable without any context information. ***The decision should be case dependant and it should be based on the prioritization of the criteria fitting into the particular scenario.*** The only exclusive inference we can make *for the time being*, is that the less suitable mean is utilization of GPUs for the reasons we have mentioned. However, it almost sure that hardware architecture changes designed by vendors for the near future can change the predicaments significantly [7]. Furthermore, various factors are related to the impact of sensoring on the target host. Some aspects are beyond the capabilities of the programmer (e.g. the algorithms forming the working sets of the caches or the cache architecture). This combination of hardware and OS solutions cannot be fully predicted. However, multiple parameters are tunable like the organization of the data structures under observation and the frequency of the observation rounds. It is highly possible that careful measuring of the system's characteristics (e.g. cache hierarchy model, cache line size etc) is critical for the afore mentioned decisions.

Table4.1: Conclusive characteristics of sensoring models

	In-host	**VMM**	**Out-host**	
			PCIe	**GPU**
Characteristics				
Full memory access	✓	✓	✓	✗
Physical security	✗	✓	✓	✓
Bootstrap checking	✓	✗	✓	✗
OS interpositioning	✓	✗	✗	✗
Processing power	✓	✓	✓	✓
Shortcomings				
TLB poisoning	probable	probable	likely	likely
Memory redirection	unlikely	probable	likely	likely
Hooking	likely	likely*	unlikely	unlikely
Message spoofing	likely	unlikely	unlikely	unlikely
Performance Impact				
Latency	low	high	medium	medium
Medium contention	medium/low**	medium	medium	high
Access pattern	low	high	medium	high

* **In this context hooking implies hijacking of registers that can result in memory remapping [70].**

** **The impact is analogous to the number of observers.**

Chapter 5

Kernel Control Flow Analysis

In the previous chapter we gave an overview of the aspects involved in data extraction in a distributed environment. At this point we are going to examine how these information should be processed and against which data it should be compared to, to reveal any malicious activity. Particularly, for an anomaly detection system to occur, a collection of legal kernel specifications must be defined. Any deviation from these specifications is examined and possibly marked as an anomaly. This includes four aspects which shape the structure of this capter. These are:

1. **Categorizing the attacks** that could be identified at run-time.
2. Formulating the **legal states** of the kernel's control flow.
3. **Analyzing** the results of the sampled data and compare them against the legal states.
4. **Report anomalies** based on the extend of the deviations for the expected values.

5.1 Kernel Violation Threats

As it is widely known, usually the step following any exploitation process is the installation of a permanent mechanism that maintaints access to a conceived machine. This is mostly done by a special category of malware known as rootkit and according to ***Petroni and Hicks*** (**2007**) an analysis of some well known rootkits reveals that there are five generic attacks: **(1) data hiding** (e.g. Synapsys rootkit) **(2) privilege escalation** (e.g. Rkit) **(3) reentry/backdoor** (e.g. Knark) **(4) reconnaissanse** (e.g. Anti Anti Sniffer) and **(5) defence neutralization** (e.g. Adore-ng).

Since nowadays, detection of rootkits existing on the user-space can be trivally detected the main consideration should be kernel-level rootkits. Most of these attacks, if not all of them, require some level of kernel subversion but there are multiple paths to do so. Examination of these paths could be based on the type of the modified object and the duration of the violation. While the latter includes only two potential states, i.e., **persistent** and **transient**, the former consists of more cases [60]. In particular, an attacker is able to perform **detour attacks** by modifying the values of multiple components affecting its behavior, i.e., kernel's .TEXT segment, registers, control flow objects (e.g. entries of the sys_call_table) and data flow objects (e.g. reducing the entropy of /dev/random [10]).

Based on the previous observations it becomes evident that the effectiveness of an anomaly-based IDS vitally relies on our ability to encapsulate the legal states of the running kernel. This is rather challenging due to its size and complexity. Once this has taken place a very carefull analysis and combination of the results taken during each observation should take place. That is not necessary when analyzing structures with static nature (e.g. the system call table whose entries are not modifiable) but it is critical for dynamic checkpoints. The latter requires a stochastic analysis of the possible transitions with use of conditional probabilities.

5.1.1 Attacks' Taxonomy

Based on the conclusion we can now give a taxonomy of the attacks that could be triggered.

1. **Control Flow Subversion**: an attacker can perform a malicious activity by manipulating the kernel in such a way that its normal execution flow is corrupted.
2. **Data Flow Subversion**: manipulation of kernel objects that do not change the normal execution flow but only the working set available for specific tasks.

As an example you can consider what is explained by ***Petroni and University of Maryland*** (**2008**). According to the elaborated example, the task of hidding a process from listing tools can be performed at least in two ways. Firstly, by changing any of the components that decide the direction of the execution (see Figure 5.1 – variation of related image by the same source). These components are:

- modification of the idtr registers,[15]
- rewriting of the functions called during the counting process inside the .TEXT segment,
- hijacking of the function pointer either in the system call table (affecting sys_getdents) or in the file structure (affecting proc_root_readdir).[16]

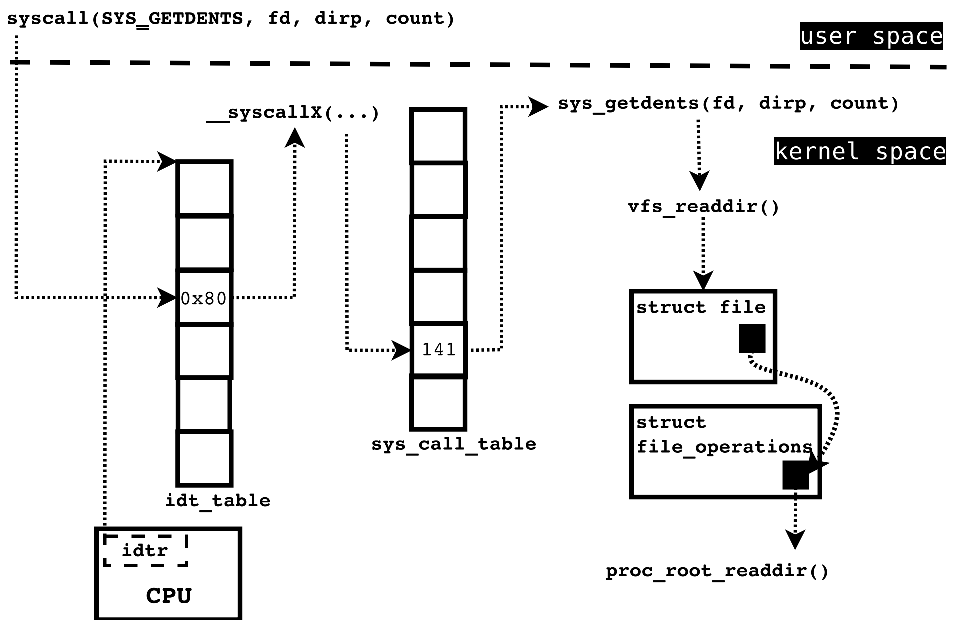


Figure5.1: Execution flow diagram for process counting through proc file system.

[15]Alternatively the **sysenter, sysexit** instructions can be used for a system call. This method involves hijacking of __kernel_vsyscall table.

[16]The last manipulation requires a sequence of accesses to perform the necessary overwrite action (file->file_operations->(*readdir)).

A second method requires manipulation of the running task list structure maintained by the linux kernel. Manipulation of the prev and next pointers of a node standing for a malicious process[17] [83], results in excluding the process from counting activities but it does not suspends its execution due to the seperates structures used by the kernel for scheduling. For an illustration look at Figure 5.2 as taken from [62].

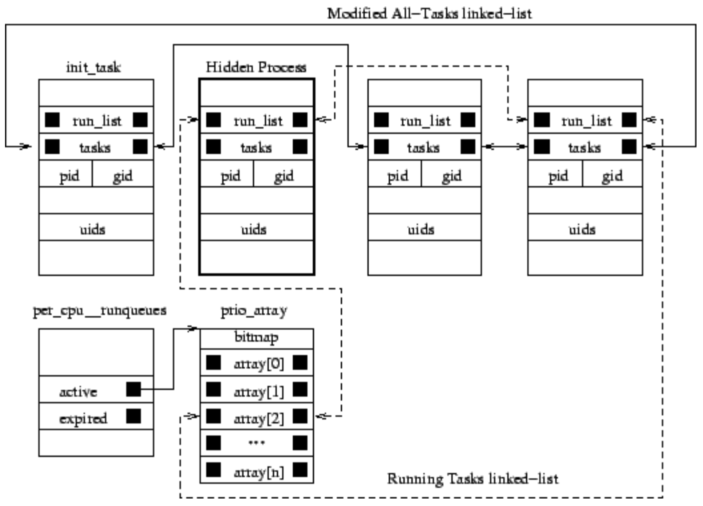


Figure5.2: Data flow process hiding.

The two possible scenarios for process hidding form persistent kernel modifications, i.e., the manipulations in the data structures are permanent. However, a transient modification for the same mission was implementented by ***Wei et al.*** (**2008**). The attacks descibed are based on soft-timer interupts [17, 44] and clearly show that it is possible for an intruder to manipulate checkpoints temporarily but still perform permanent malicious activities in the victim host (e.g. process hidding, key-logging etc).

As a result a detection mechanism working on state-based control flow integrity should both look for *atomic* and *cross-sectional* observations [46]. By the term *atomic* we mean that detection of a subversion attempt can be done by observing a single component (e.g. potential function pointers that might be overwritten – consider example 1). On the other side detection of data flow manipulation requires cross observations, i.e., comparison between various data structures (e.g. the number of nodes in the task list should match the number of nodes in the scheduling lists maintained by the kernel).

5.2 Formulating States

To achieve the kernel states' formulation one should give a semantic representation of the kernel and the monitoring components involved. A complete representation of the symbols that will be used throughout this chapter can be found in table 5.1.

5.2.1 Terminology

We consider a kernel as an automaton $\mathcal{K}$ for which we are able to form an approximation $\mathcal{K}_\alpha$. $\mathcal{K}_a$ consists of data elements, m of which are under surveillance by the monitor ($\mathcal{D} = \bigcup_{1<i<m} d_i$).

[17]using the REMOVE_LINKS for example

Each *semantic checkpoint* comprises of one or more states s where $\mathcal{S}_{d_i}$ is the set of all possible states for a data element ($\mathcal{S}_{d_i} = \bigcup\limits_{s \in \mathcal{S}_{di}} d_i^s$). Static elements are those consisting of one state exactly (e.g. an entry of IDT or sys_call_table) while dynamic elements pass through more states. The corpus of possible states for a data element form a **partially ordered set** because transitions between states are not arbitrary [45]. Due to this ordering we can establish a relation between two states so that a state s of element d_i "happens before" another state $s+1$ ($d_i^s \rightarrow d_i^{s+1}$). If the transition between the two states is possible to happen with probability p then we have a "potentially happens before" relationship ($d_i^s \overset{p}{\hookrightarrow} d_i^{s+1}$) [45]. Furthermore, in concurrent observations it is possible to have causalities between multiple data elements (e.g. the running task list and the scheduling lists). These relations fall into three categories [47]:

- **strong causality**: if a transtion of a data element from state s to $s+1$ **will** happen due to **unique** condition,
- **weak causality**: if a transtion of a data element from state s to $s+1$ **will** happen due to a **set of possible** conditions,
- **conditional dependency**: if a transtion of a data element from state s to $s+1$ **might** happen due to a **set of possible** conditions.

Graphically each state can be represented by a node, with the directed labeled edges standing for possible transitions with the related probabilities. As an example consider image 5.3. Apart from the illustrated representation we can have a more intuitive understanding of the causal relations between the states [31]. For example we see a "happens before" relation exists between nodes d_j^s and d_j^{s+1} while a "potential happens before" between $d_i^{s'}$ and d_i^{s+2}. Examples of causalities are:

- **strong causality**: $d_i^s \underset{d_i^{s-1}}{\overset{q_1}{\rightarrow}} d_i^{s+1}$,
- **weak causality**: $d_i^{s'} \underset{d_j^{s-1}+d_j^{s-1'}}{\overset{q'}{\hookrightarrow}} d_i^{s+2}$ and
- **conditional dependency** $d_i^{s+1} \underset{d_j^s}{\overset{q_2''}{\rightsquigarrow}} d_i^{s+3}$.

Finally, it is worth noting that we can also calculate the probability of falling in a state through a particular path [45]. For example, the probability of going into state d_i^{s+2} is p', p'' or $p \cdot q_2$. As we will see this is a critical aspect for a monitoring system that takes into account the transistions between the states.

5.2.2 Pitfalls

The synopsis and the explanation provided above might mislead somehow the reader upon the simplicity of such a process. The recent advances in hardware hyperthreading technologies affected the kernel design and this can be clearly infered by observing the changes in the Linux kernel from version 3.2 and so. Addition of more threads to utilize advanced capabilities of hardware affects the kernel state concurrency. This aspects makes it is rather difficult to formalize the behavior of core elements, which is a vital aspect because a transition sensitive analysis cares about the **serialization** of status changeovers. In an environment characterized by concurrency the result is a ***probabilistic interleaving of operations*** [46] and as such there is a likelihood of not being able to decide the order of the state switching.

Table5.1: Notation overview for kernel state-based representation

Symbol	**Explanation**
Symbolic Primitives	
$\mathcal{K}$	OS kernel automaton
$\mathcal{K}'$	modified OS kernel automaton
$\mathcal{K}_\alpha$	kernel's approximation
$\mathcal{P}^N$	corpus of N observers
P_i	observer i
$\delta_\mathcal{O}$	interval between two observations
$\mathcal{D}$	group of observed data elements (**semantic checkpoints**)
d_i	data element i in $\mathcal{D}$ ($d_i \in \mathcal{D}$)
d_i^s	data element i at state s
$\mathcal{S}_{d_i}$	set of possible states for data element d_i, $\mathcal{S}_{d_i} = \bigcup_{s \in \mathcal{S}_{d\,i}} d_i^s$
$\mathcal{S}_\mathcal{D}$	set of possible states for each data element d_i, $\mathcal{S}_\mathcal{D} = \bigcup_{d_i \in \mathcal{D}} \mathcal{S}_{d_i}$
Relational Primitives	
$d_i^s \rightarrow d_i^{s+1}$	happens before relation between state s and $s+1$ for data element d_i
$d_i^s \overset{p}{\hookrightarrow} d_i^{s+1}$	potentialy happens before relation between state s and $s+1$ for data element d_i
$\delta_{d_i^s \rightarrow d_i^{s+1}}$	transition period from state s to $s+1$ for data element d_i
$p_{d_i^s \rightarrow d_i^{s+1}}$	probability of transition from state s to $s+1$ for data element d_i
Causal Primitives	
$d_i^s \underset{d_j^{s'}}{\overset{p}{\rightarrow}} d_i^{s+1}$	data element d_i will fall to transition from state s to $s+1$ only if data element d_j falls into state s' (**strong causality**)
$d_i^s \underset{d_j^{s'}+c}{\overset{p}{\hookrightarrow}} d_i^{s+1}$	data element d_i will fall to transition from state s to $s+1$ i f data element d_j falls into state s' or some other condition c occurs (**weak causality**)
$d_i^s \underset{d_j^{s'}}{\overset{p}{\rightsquigarrow}} d_i^{s+1}$	data element d_i may fall to transition from state s to $s+1$ if data element d_j falls into state s' (**conditional dependency**)

5.2.3 Monitoring Models

Based on the previous specifications we will understand that there are two ways to design a monitoring system. The first is **state-based** (or transition-ignorant) monitoring and the second one is **transition-sensitive** (or transition-dependent). Each of the methods offers a number of advantages, mainly between the standard trade-off among security, complexity and performance. A state-based system, after every observation round, checks whether any of the sampled states of a data element $d_i^s \notin \mathcal{S}_{d_i}$. For regions whose not only the value but also the location is known a priori (e.g. the .TEXT segment) cryptographic primitives like hash functions could be used for the entire locus. For objects whose value is immutable but transportable after every reboot they should be relocated [12, 43, 62]. At this point we should stress the fact that most well known rootkits perform persistent amendements that divert the control flow of the kernel mainly because the target is the existance of a long-term surreptitious functinality [60].

On the contrary a transition-sensitive monitor attempts to correlate results from various observation

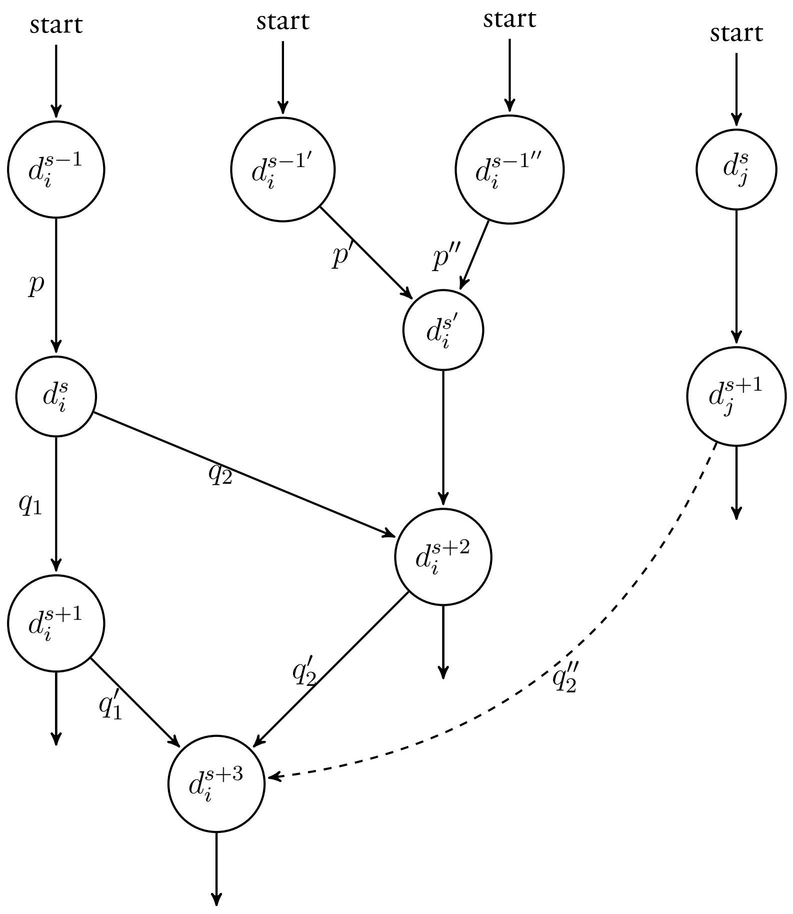


Figure5.3: Graphic representation for kernel state transitions.

rounds. That makes it possible to inspect the behavior of dynamic elements through a probabilistic analysis. Such a stateful rationale tries not only to check whether a captured state $d_i^s \in \mathcal{S}_{d_i}$ but also to detect violations of "(potential) happens before" relations or states that are *not likely to happen.* Although there are a few assumptions that should be revised when designing transitional analysis as mentioned by ***McEvoy and Wolthusen*** (**2008**).

Firstly, it should be noted that in transitional observations it is possible between two observations to have some missing transitions if $\delta_O > \delta_{d_i^s \rightarrow d_i^{s+1}}$.[18] That implies that the inspector might have to examine multiple potential execution paths to decide whether a transition from one state to another is possible or not (e.g. based on the previous graph transition from $d_i^{s'}$ to d_i^{s+1} implies contol flow subversion but transition from $d_i^{s'}$ to d_i^{s+3} is possible if the inbetween transition was not observed).

In addition if there are more than one processes performing sampling during a round of observation it might happen that different states are captured by the observers. Due to the interleaving nature of parallel operations there is no baseline to decide the actual order with which they were performed. These two aspects are critical for **minimizing false positive & false negative alarms** because the after effect of subversion is measured. As a result we might have inconsistent measurements that arise either due to a malicious activity or because our measurement is problematic.

[18]For this reason when the kernel control flow analysis is performed, it might be useful to measure the number of clock cycles needed for certain transitions (e.g. using inlined assembly with **rdtsc** instruction).

5.2.4 Kernel's Observability

At this point we should stress that problematic measurements are not necessarily a result of bad analysis. On the contrary, on complex systems, like the kernel of an OS, we are not able to perform conventional control flow analyis, which means that we cannot obtain a full view of the system by observing its output. In mathematic theory this property is known as **observability** and in systems that do not fulfill this property we observe a selected subset of elements ($\mathcal{K}_a$). The problem is to "*identify the minimum set of sensors from whose measurements we can determine all other state variables*" [42]. This **partial observability** of the system dictates the existance of a sensoring model tailored to work on this abstraction layer without confusing our *believed* set of states with the real ones [14]. For reasons of feasibility it would be a reasonable assumption to say that, such a monitor cannot be tuned do deal with dynamically loaded kernel modules, because these change the control flow of the kernel and there is no safe way to predict their impact.

5.2.5 Subtle State Fuzzing

The main characteristic of a transistion-sensitive anomaly-based IDS is the **stochastic prediction of potential inconsistencies**. As we discribed discrepancies might arise both from "alien states" [47] and from "out-of-order execution" because Besides the introduction of illegal kernel data, malicious kernel modules can also utilize legal kernel code for illicit purposes, by performing arbitrary jumps or return operations [79]. Detection of the latter is a rather challenging task because to detect this case one should have the ability to order the observed executions. This is not possible however, in multi-processing environments where the lack of sensors located at different processing units to share a unique clock creates the physical inability to use a time-stamping mechanism for the collected events [31].

That speculative nature of state sorting algorithms, could be exploited by an attacker by causing malicious actions resulting in out of order execution of legal states that could not be fully marked as malicious because there is no good way to decide the precedence between the sampled states. The ability of an intruder to perform such a task with success relies at some point to his knowledge of the sampling strategy. It should be noted that this teleologic aspect of this research area could easily be characterized as a poorly studied case. This gap analysis should be further examined because it generates multiple challenges.

In addition, the interpositioning of the executed instructions between the kernel code and the sensors (even in a uni-processing systems), deprive the ability to have a full view of the execution flow. Measurements taking place on the same systems twice will definately produce different results for this reason because we cannot control the alternating order of the execution between multiple tasks. Apart from the two above mentioned aspects, we should also bear in mind the possibility of missed transitions between two rounds of observation. This gives an attacker the ability to perform changeovers that are not direct but are within the threshold of probabilistic acceptance by a stochastic sensor. At this point we should not forget that every IDS aims to minimize false indications and hence to strike a balance between false positives and false negatives. Augmenting the fragility of an observer implies increase of the probability to produce false alarms.

5.3 Practical Considerations

Before closing this chapter, it is worth noting some aspects that come along with the above mentioned. Every monitor analyzing the kernel state specification and inspection should be able to locate the critical memory regions. In Linux systems the first available source for such information is the System.map symbol table. This is mainly used by the kernel and provides a mapping of kernel symbols and the related virtual memory addresses. In addition it gives a sign about the memory region in which each symbol is located and it is extremely useful for kernel debugging purposes.

5.3.1 Locating and Extracting Symbols

In our context System.map should firstly be analyzed to reveal which symbols are located in the same address after rebooting the machine and which not and at the same time categorize at which memory region each symbol resides in. These symbols can be used as root nodes and through recursive traversal we can emerge the the values of the symbols we are interested in (e.g. function pointers). Once the invariants are distinguished then locating comes in turn. The virtual addresses found in the same file can be used for locating purposes and an example of a root symbol could be **super_blocks** which is vital for inspection of elements related to the /proc virtual file system [11]. The difficulty faced by an extractor algorithm is the distinction between useful pointers and pointers of on interest [12, 60]. The reason is the weak type systems supported by C (e.g. generic pointers – **(void *)**) or the capability to perform polymorphic inheritance in C.[19]. Once the objects in question are traversed the legal states and transitions can be examined and complete kernel's state legal modeling.

For very special types of stealthy but powerful attacks (e.g. soft-timer driven attacks [84]) it might be necessary during the static analysis of the code to generate a "dictionary" of the legitimate assignements. According to the strategy described in the above publication a symbolic representation of the symbols extracted through source code parsing is maintained and located at run-time. The reason is at compile time it is possible not to know the addresses because the interrupt might also be used by dynamically loaded kernel modules.

5.3.2 Making Inferences

At run-time the traversal should take place at every round of observation on objects residing in the heap and stack of the host system. It is possible to acquire inconsistencies due to the incomplete state in which the kernel was captured. If for example some allocations of memory regions were not completed, we might end-up examining not valid memory locations. To ensure the opposite one could examine the page tables of the target and explore if the memory region in question is valid or not. An analysis of the page table structures for Linux 64-bit architectures can be found in [51, Chapter 4].

5.3.3 Reducing Indeterminism

Modern systems offer various techniques that increase the efficiency of running programms by utilizing out-of-order execution that reduce idling intervals of threads, processes' preemption etc. When we analyze the behavior of the kernel control flow, it is sometimes useful to perform measurements related to the clock cycles needed for special tasks. We will elaborate on this in the next chapter where we explain the challenges we faced during the development of a LKM for this reason. Here we briefly mention that elimination of this indeterminism and when developing kernel measurement tools special consideration is needed to achieve such a reduction. Some of these aspects include configuration of non maskable interrupt handlers and manipulation of the platform performance counter for throttling balancing because the **rdtsc** instruction used for clock cycles measurement is affected by power saving which can be disabled through the BIOS.

[19]For the interesting reader the link below is a useful source of information http://modal-echoes.blogspot.co.uk/2007/03/implementing-polymorphism-in-c.html

Chapter 6

Case Study

The preceding analysis clearly illustrates that there are numerous paths involving extensive research. In this chapter we attempt to go into some further details on one crucial aspect: a more granular investigation of performance evaluation criteria and how could we perform cost improvements that could give us useful inferences by precise measurements through cache programming hints.

6.1 Observation Cost in more Depth

Assisted by existing cost models, the performance degradation imposed by applying memory inspection for intrusion detection can be determined. Bearing in mind that, not only the frequency of observations, but also its organization of the observed data in terms of *size*, *mode* and *balancing*[20], an approximated performance loss of systems under investigation can be computed. In this section we will try to explore in more depth some aspects of the observation cost model caused by the caches. The problem is rather challenging for various reasons. Firstly, measuring the behavior of cache memory is a rather demanding task in many ways:

1. Hardware manufacturers provide no information about the internal organization of cache memory, apart from some rudimentary elements [50, 57].
2. Cache behavior is affected by factors not 100% examinable like hardware internal management (e.g. working set grouping), or OS intermingled approaches like prefetching.
3. Tools offered for such missions (like VTune by intel [64]) might suffer from outliers that cast the original performance of the system. The outliers can be caused by many sakes of indeterminism like the workload of the running system during the experiments, preemption of processes by the OS, hardware & software interrupts, turbo mode, clock frequency scaling, hyper-threading etc.

Secondly, any data exctracted from conducted experiments will require some careful statistical analysis to offer safe and useful conclusions. Furthermore, for the results to be more generic, analysis on various platforms with different computer architecture is necessary. The latter is one of the problems we face while this project is under development.

6.1.1 Overview & Countermeasures

Tuning the organization of a program at a source code level to tackle cache issues, is a rather demanding aspcet composed of complicated programming principles. These can help in reducing general

[20]Or more accuretely distribution policy across multiple processing units

performace loss through the runs of the system. In our case we will specifically try to see how observed data structures can be restructured to reduce the penalties from the observations. A good description of the generic problem were discribed by ***Sears*** (**2000**). Briefly cache memory issues can be grouped as follows:

- **Thrasing**: term describing increased competition for cache lines from different threads causing various abnormalities:
  - **False sharing**: as we mentioned in previous chapter distribution of a cache line among various cores which intensively modify and read its contents in parallel. This causes a forced memory update every time a cache line is invalidated (see Figure 6.1 as taken from [3]). These conditions increase coordination frequency & degrade performance and programmers can identify and reduce them [3].
  - **Cache pollution**: exists when prefetched data evicts heavily used cache lines that afterwards need to be reloaded.
  - **Cache footprint**: the measure expresses the amount of cache that contributes to the performance of a task. Increased prefetching of not heavily used lines by a program, reduces the footprint and results in cache pollution.

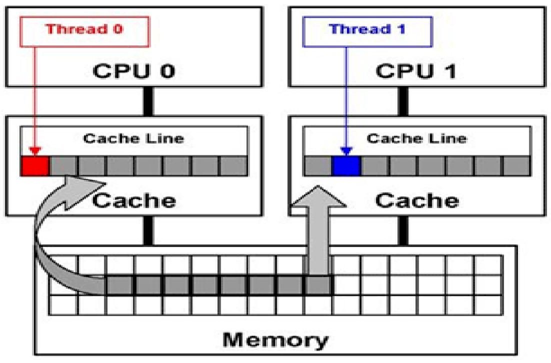


Figure6.1: Illustrated view of false sharing.

- **Layout**: the organization of a program's memory layout along with the rationale of memory handling algorithms decide the effect of the previously mentioned issues. Strategies that could be employed to reduce the problem are:
  - **Grouping**: As we explained in Chapter 4 it is rather useful to group the various data elements under observation so as to avoid intermingling of observed and unobserved variables. This is to reduce the cache coordiation every time the sensor polls memory regions that contain unmodified semantic checkpoints that coexist with modified elements of no interest in the same cache line. For that we want to compact observed elements in as few cache lines as possible and seperate them from other irrelevant to our activities members by spreading the latter out. "***If the frequently accessed fields are collected into a single cache line, they can be loaded with a single memory access. This can reduce latency and cache footprint.***" [71]
  - **Alignment**: Although compilers employ their own techniques to perform automatic bounday aligning of variables, the programmer can use asigned attributes and padding as we will see afterwards [4]. [21]

[21]We mainly care for arrays and C `struct` variables that encapsulate the most critical kernel chechpoints under observation.

- **Packing**: For reasons related to the previous item, it might be useful to perform packing of variables. Packing changes the alignment policy of the compiler and consumes as less memory as possible for a given object. It works as a contrapositive of alignment by removing as much identifiable redundancy as possible. That might be a useful tool in caches of overlapped cache elements which is preferable to seperate by narrowing down their boundaries instead of ultra-padding.[22]

Techniques related to the reduction of the above mentioned issues can be very useful due to the typical size of potentially observed data structures (e.g. super_block, super_operations, inode, inode_operations etc.), which exceed the typical size of a cache line (32–64 bytes) and where not every single element is of the same importance for the sensors (i.e. some of them might not be observed at all). This along with the massive amount of observations can reliably lead as to the conclusion that examination of such issues can result in increased performance for an observation mechanism.

As we said previously, for somebody to perform such a scrutinized analysis with high presicion the utilization of standard measurement tools like VTune is not enough. For this reason the developement of a program working at kernel level is necessary. In this way, many aspects of indeterminism can be eliminated in the observed host and for this reason ***Paoloni*** (**2010**) developed a detailed white paper with a dedicated LKM [54]. However, this source is not enough because it does not include any implementation causing cache synchronization runs and there are various improvements that could be done. For example, the memory management needs some improvements and the number of iterations run are affordable by big platforms only, causing kernel hangs. For this reason in section A.2, we will provide a changed version of the module with some additional observations that could help in conducting the future work connected to this aspect. We hope that this practical approach will provide the baseline for useful experiments that to the best of our knowledge are not yet developed. It should be noted that kernel programming poses many obstacles not present in userspace programs. For instance, distribution of threads in a LKM across various units is not as simple as illustrated in section A.1.1 and the reason is that libc is not visible by the kernel and the APIs provided cannot be utilized. Instead, the kernel internal functions should be used to set the affinity of each thread and synchronize them.

[22]Packing is not the default behavior of compilers because aligning improves run-time efficiency (e.g. use of ldd and std for double loading and store respectively, works faster).

Chapter 7

Future Work

This project covers a big research spectrum on anomaly-based host IDSs and for this reason a more fragmented analysis of the various dimensions overviewed is a vital necessity. Although a complete implementation of an observer would be the ideal scenario, we would suggest some more specialized efforts. In our future work we first of all intent to work on what we consider the biggest analysis gap, which is sensoring fuzzing. To do so we would suggest the investigation of legal execution paths that can result in performing malicious activity if executed out of order. This is because it is mentioned as an assumption in various publications but to the best of our knowledge there is no real-world scenario documented. It would also be highly valuable to observe the fluctuation between various processor clocks by implementing a small assembly snippet that observes a particular memory area from different units. The reason we refer to assembly snippets is the need to reduce "noise" introduced by programming APIs like system calls etc. If every observer stamps the polled results using a centralized clock using a waiting queue strategy, it would the be easier for a defender to enhance the viability of event ordering.

The second part worth of consideration is the devopment of some LKM sampler that measures the cost of cache coherency protocol runs. That requires the development of a benchmark working on kernel level (possibly based on what can be found in A.2) by creating threads working on different cores. It would be highly advisable to perform the expirements on a system with more that one processors and not only cores where the delays caused by cache synchronization are higher and because all the cache levels are not shared. The threads simulating the observed host and the observers should work an a shared data structure of the necessary size (512 bytes is a good start) and the improvement caused by grouping nested variables shall be measured in terms of clock cycles. Based on the inferred results, it would be useful to come up with some revised layout for a few primitive kernel data structures of the Linux kernel.

Another interesting aspect to investigate is the ability of an attacker to use a private cache levels to hide internals of its behavior as explained in subsection 3.3.2. For this reason, we consider rather challenging the examination of the behavior of various platforms on this part. Although on Intel architectures we did not manage to find any way to implement cache line locking apart from contiguous prefetching of certain pages [23], the findings about other architectures are rather promising. ARM architectures for example include real time systems designed years ago, to offer well-defined run-time behavior[24]. In addition, SH-4 RISC CPU includes some cache control instructions like `movca.l` which reserves a cache line and does not even synchronize it with RAM. Apart from the architectural examination, it is worth of investigation to implement a simple obfuscation algorithm on a standard architecture (e.g. Intel) by trying to keep the related semantics private using prefetching, and reporting the percentage of successful runs.

[23]However, prefetching does not force the OS to load a page on the cache but it offers high possibility of success. If the OS workload forces high use of cache memory, then it is possible to ignore prefetching demand by userspace processes.

[24]http://sourceware.org/sid/component-docs/hw-cache.htmlbehavior-line%20locking

Moreover, it would be highly useful to examine the feasibility of many attacks on memory acquisition mechanisms as discussed in Chapter 4. Many of the attacks enumerated are documented but never found on real world malware. Based on the inference, one could examine if for example shadow walker rootkit could be implemented for multi-processing systems and if yes which semantics of the kernel are violated so that an anomaly-detector could detect the existance of the attack. The same holds for blue-pill like rootkits and generally speaking it would be very profitable in terms of research effort to come up with some conclusions related to the feasibility of such attacks. The reason is that for a rootkit performing so advanced attacks, it is necessary to exist in kernel space, which lets a window of detection always possible. Nonetheless, during our literature review we identified various assumptions that do not hold any more in many cases. For example, some real world detectors (e.g. [12, 61]) use interfaces for kernel memory access (e.g. /dev/kmem) which are not enabled any more by default in the Linux kernel. This is in full sync with our initial observation that this project is a snapshot of the current-state-of-art and as such it would be worth of investigation to examine alternative interfaces and potential subversion of those.

Last but not least, there is no doubt that anomaly detectors provide a good way to report abnormalities but compared to signature matchers, they are less radical in terms of reaction. For this reason, it is of high importance to examine ways with which the sensors could go a step further to react in such actions by detecting malicious processes and explore their behavior. Isolated sensors are less powerful at this point because they use more abstract interfaces for communication with the host OS. In addition, sensors could be designed in such a way that they could restore kernel's legal states (e.g. a hijacked function pointer) but this could possibly reduce the ability to trace illicit activities.

Chapter 8

Conclusions

Current systems dedicated to host intrusion detection are mainly based on signature matching techniques. Their impact on the observed system is high in terms of performance degradation and they rely on the ability to know an identification characterizing the threat causing the subversion of the system. Consequently, an attacker can employ modern methods of concealment and benefit from the non-deterministic behavior of concurent systems, can neglect the ability of such an IDS to detect a novel attack vector.

Ideally, an intrusion detection mechanism would be a system able to detect malicious fanctionality without any knowledge of the attack gear. This reality forces the exploration of alternative intrusion detection mechanisms like behavioural or anomaly based. Chapter 2 presented both the limitations mentioned and an overview of proposed approaches that can adopt an anomaly based IDS within a multi-processing system. Such a system focuses on detection of symptoms that consitute usual consistencies of a kernel-level attack. Namely, these symptoms are subversion of the OS kernel.

The present work examined the capabilities and limitations of anomaly-based IDS based on concurrent memory inspection of critical kernel data structures. The investigation of the sensoring models – be it implemented on embedded or auxiliary resources – at Chapter 4, concludes that no model fulfills all the necessary security & performance properties. Therefore, we examined both the potential attacks and limitations along with the cost model that accopanies each and every of them. Nevertheless, the data acquisition should be followed by a formalized representation of the results gathered and an analysis of the host system to represent normal kernel code execution. That was the objective of Chapter 5 and it provides an exhibition of the usefulness and importance of the predetermined properties that encapsulate the security violations that indicate malicious actions.

A more detailed analysis of the aspects related to the cost of observation was performed on Chapter 6. The challenge of this task is the necessity to reduce noise during measurements which is added by both the hardware and the OS. Measuring the clock cycles needed for particular activities requires bypassing normal interfaces provided by corporate software and working at kernel level. Only under these conditions we can have reliable results that can resemble optimizations gained by reducing sources of delay like false sharing.

Last but not least, most of the conclusions made in this project are prone to changes since the evaluation is based on the current state-of-the-art. Changes of hardware architecture and software engineering desing of OS kernels should be closely attended as the can change the composition of the investigated components. Analysts and researchers must stay tuned with these changes before using conclusions exported from this project.

Appendix A

Source Code

A.1 Samples

A.1.1 Thread Distribution

In this program, the number of cores running on the host system is detected, and according to this number, a distribution policy for the four running threads, excluding the main thread, is decided. Once execution is finished, a number of messages is printed to show the cpu sets where each thread can run on. Do it by using the non portable function pthread_setaffinity_np(). This requires the prior definition of one or more cpu sets, through the cpu_set_t data structure as defined in the sched.h library. The directives defined in the same library can be used for the formulation of the cpu sets. If the POSIX threads API is not supported use sched_setaffinity() but instead of passing the pid of the process, specify value returned by gettid() or getpid() for the main thread.

Listing A.1: Thread distribution to multiple cores using pthread API

```
/**
 * Compile: gcc -o dthreads dthreads.c -lpthread
 * Non portable functions are used. The program runs
 * only on certain Linux Platforms.
 */

#define _GNU_SOURCE
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<errno.h>
#include<sched.h>
#include<unistd.h>
#include<stdint.h>

#define handle_error_en(en, msg) \
    do {errno = en; perror(msg); exit(EXIT_FAILURE);} while↩
       (0)

#define DOUBLE_CORE     0x0002
#define QUATRO_CORE     0x0004
```

```
typedef struct _NCores{
    long ncores;
    long max_ncores;
}NCores;

int countcores(NCores*);
void NCores_init(NCores*);

void NCores_init(NCores *nCores)
{
    nCores->ncores = -1;
    nCores->max_ncores = -1;
}

uint count_cores(NCores *nCores)
{
    nCores->ncores = get_nprocs_conf();
    nCores->max_ncores = get_nprocs();

    return (uint32_t) nCores->ncores;
}

void set_cpu_sets(NCores *nCores, cpu_set_t *cpuA, cpu_set_t ↩
   *cpuB, cpu_set_t *cpuC, cpu_set_t *cpuD)
{
    int i;

    for (i = 0; i < nCores->ncores; i++)
    {
        if ((i%4) == 0)
            CPU_SET(i, cpuA);
        if ((i%4) == 1)
            CPU_SET(i, cpuB);
        if ((i%4) == 2)
            CPU_SET(i, cpuC);
        if ((i%4) == 3)
            CPU_SET(i, cpuD);
    }
}

static void *pthread_initiallizer(void *arg)
{
    sleep(8);
    int i, temp;
    char *message = (char *) arg;
    cpu_set_t cpuX;

    CPU_ZERO(&cpuX);
    temp = pthread_getaffinity_np(pthread_self(), sizeof(↩
        cpu_set_t), &cpuX);
    if (temp)
        handle_error_en(temp, "pthread_getaffinity_np");
    printf("\n%s can run on cores: ", message);
    for(i = 0; i < CPU_SETSIZE; i++)
        if (CPU_ISSET(i, &cpuX))
            printf("%d ", i);
    printf("\n");
}

int main(int argc, char *argv[])
{
    int i, j, s, temp, dual_core, quat_core, mult_core;
    uint32_t cores_flag;
    cpu_set_t cpuA, cpuB, cpuC, cpuD;
    pthread_t thread_1, thread_2, thread_3, thread_4;
    const char *arg1 = "1st thread";
    const char *arg2 = "2nd thread";
    const char *arg3 = "3rd thread";
    const char *arg4 = "4th thread";
    NCores nCores;

    NCores_init(&nCores);
    cores_flag = count_cores(&nCores);
    printf("cores = %u\n", cores_flag);

    sleep(1);
    s = pthread_create(&thread_1, NULL, &pthread_initiallizer↩
        , (void *) arg1);
    if (s)
        handle_error_en(s, "pthread_create");

    sleep(1);
    s = pthread_create(&thread_2, NULL, &pthread_initiallizer↩
        , (void *) arg2);
    if (s)
        handle_error_en(s, "pthread_create");

    sleep(1);
    s = pthread_create(&thread_3, NULL, &pthread_initiallizer↩
        , (void *) arg3);
    if (s)
        handle_error_en(s, "pthread_create");

    sleep(1);
    s = pthread_create(&thread_4, NULL, &pthread_initiallizer↩
        , (void *) arg4);
    if (s)
        handle_error_en(s, "pthread_create");

    CPU_ZERO(&cpuA);
    CPU_ZERO(&cpuB);
    CPU_ZERO(&cpuC);
    CPU_ZERO(&cpuD);
```

```
set_cpu_sets(&nCores, &cpuA, &cpuB, &cpuC, &cpuD);
/* decide system type and ensure that all cpu_set_t(s) ↩
   are initialized properly */
dual_core = (!(cores_flag ^ DOUBLE_CORE)) && CPU_COUNT(&↩
   cpuA) && CPU_COUNT(&cpuB);
quat_core = (!(cores_flag ^ QUATRO_CORE))  && CPU_COUNT(&↩
   cpuA) && CPU_COUNT(&cpuB)
    && CPU_COUNT(&cpuC) && CPU_COUNT(&cpuD);
mult_core = (cores_flag >= 8) && CPU_COUNT(&cpuA) && ↩
   CPU_COUNT(&cpuB)
    && CPU_COUNT(&cpuC) && CPU_COUNT(&cpuD);

if (dual_core)
{
    temp = pthread_setaffinity_np(thread_1, sizeof(↩
       cpu_set_t), &cpuA);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_2, sizeof(↩
       cpu_set_t), &cpuA);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_3, sizeof(↩
       cpu_set_t), &cpuB);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_4, sizeof(↩
       cpu_set_t), &cpuB);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
}
else if (quat_core || mult_core)
{
    temp = pthread_setaffinity_np(thread_1, sizeof(↩
       cpu_set_t), &cpuA);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_2, sizeof(↩
       cpu_set_t), &cpuB);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_3, sizeof(↩
       cpu_set_t), &cpuC);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
    temp = pthread_setaffinity_np(thread_4, sizeof(↩
       cpu_set_t), &cpuD);
    if (temp)
        handle_error_en(temp, "pthread_setaffinity_np");
}

pthread_join(thread_1, NULL);
```

```
    pthread_join(thread_2, NULL);
    pthread_join(thread_3, NULL);
    pthread_join(thread_4, NULL);

    return 0;
}
```

A.1.2 Non Executable Paths

This program is a simple illustration of the ability to create non executed paths that are not imprinted by static analysis. Running of the program shows that certain printf() statement will never be executed. The scenario could be furtherly developed by establishing a second slave thread that would subtract by 1 the condition variable and hence making prediction of the executed paths even more difficult. That is because we are not able to predict which thread would acquire first the lock of the condition variable and as a result what the order of execution will be.

Listing A.2: Illustrating unexecuted paths by joining multiple threads

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define handle_error_en(en, msg) \
    do { errno = en; perror(msg); exit(EXIT_FAILURE); } while↩
        (0)

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

#define THREAD_ALIVE          0
#define THREAD_TERMINATED     1
#define THREAD_JOINED         2

static pthread_cond_t slave_found = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

static int slaves_tot = 0;
static int slaves_act = 0;
static int slaves_ter = 0;

typedef struct _ThreadData{
    pthread_t id;
    int state;
    int sleep_time;
    char *message;
}ThreadData;

ThreadData *threadData = NULL;
```

```
static void *start_slave(void *arg)
{
    int id = (int) arg;
    int s;

    sleep(threadData[id].sleep_time);
    printf("Thread %d terminating\n", id);

    s = pthread_mutex_lock(&mutex);
    if (s)
        handle_error_en(s, "pthread_mutex_lock");

    slaves_ter++;
    threadData[id].state = THREAD_TERMINATED;

    s = pthread_mutex_unlock(&mutex);
    if (s)
        handle_error_en(s, "pthread_mutex_unlock");
    s = pthread_cond_signal(&slave_found);
    if (s)
        handle_error_en(s, "pthread_cond_signal");

    return NULL;
}

int main(int argc, char *argv[])
{
    int s, thread_id;

    threadData = calloc(argc, sizeof(*threadData));

    if (!threadData)
        handle_error("calloc");

    for (thread_id = 0; thread_id < argc; thread_id++)
    {
        threadData[thread_id].sleep_time = argc;
        threadData[thread_id].state = THREAD_ALIVE;
        s = pthread_create(&threadData[thread_id].id, NULL, ↩
           start_slave, (void *) thread_id);
        if (s)
            handle_error_en(s, "pthread_create");
    }

    slaves_tot = argc;
    slaves_act = argc;

    while (slaves_act)
    {
        if (slaves_ter == 0)
        {
            printf("first if\n");
            s = pthread_cond_wait(&slave_found, &mutex);
            if (s)
                handle_error_en(s, "pthread_cond_wait");
            continue;
        }
        else
        {
            printf("first else\n");
        }

        if (slaves_ter == 0)
        {
            printf("second if\n");
            s = pthread_cond_wait(&slave_found, &mutex);
            if (s)
                handle_error_en(s, "pthread_cond_wait");
        }
        else
        {
            printf("second else\n");
        }

        for (thread_id = 0; thread_id < slaves_tot; thread_id↩
            ++)
        {
            if (threadData[thread_id].state == ↩
                THREAD_TERMINATED)
            {
                s = pthread_join(threadData[thread_id].id, ↩
                    NULL);
                if (s)
                    handle_error_en(s, "pthread_join");

                threadData[thread_id].state = THREAD_JOINED;
                slaves_act--;
                slaves_ter--;
            }
        }

        s = pthread_mutex_unlock(&mutex);
        if (s)
            handle_error_en(s, "pthread_mutex_unlock");
    }

    printf("slaves_ter = %d and slaves_act = %d\n", ↩
        slaves_ter, slaves_act);
    return 0;
}
```

A.2 Kernel Level Measurement Tool

The LKM listed below is based on the related work by *Paoloni* (**2010**) but it includes with some variations related to the way the memory management is handled. No changes in the statistical analysis have taken place. Its purpose is to work as a baseline benchmark for future work that aims to perform very precise measurements. As a benchmarking scheme we would suggest the use of a code snippet that distributes some threads across different units. In kernel space it can be done by using the nr_cpu_ids table and kthread_creade() & kthread_bind() functions for creation and affinity assignment respectively. This threads could then work on a piece of shared data (e.g. a C struct) by emulating the reader and the writer. We can then programmatically change the format of the shared object based on the principles discussed in Chapter 6 and compare the results exported from various rounds. That could give a descent view of the potential improvements. These functions are defined in kernel/kthread.c. Within the Makefile, we also have the ability to include commands for throttling deactivation and other sources of indeterminism.

Listing A.3: Variation of well known LKM benchmarking model

```
#include<linux/module.h>
#include<linux/vmalloc.h>
#include<linux/kernel.h>
#include<linux/kthread.h>
#include<linux/init.h>
#include<linux/hardirq.h>
#include<linux/preempt.h>
#include<linux/sched.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Christos Tsopokis");
MODULE_DESCRIPTION("CPU Cycle Counter");

#define NO_OF_ENSEMBLES          5
#define SAMPLES_PER_ENSEMBLE     10

#ifdef __i386__
#define RDTSC_CLOBBERED_REGISTERS "%eax", "%ebx", "%ecx", "%↩
    edx"
#define SUPPORTED_REG              "%%eax"
#elif __x86_64__
#define RDTSC_CLOBBERED_REGISTERS "%rax", "%rbx", "%rcx", "%↩
    rdx"
#define SUPPORTED_REG              "%%rax"
#else
#error unknown platform
#endif

#define CLOCK_COUNT_INIT(initialization) ↩
                                                                \
    do { ↩

        \
        register unsigned init_cycles_high_bits, ↩
            init_cycles_low_bits;                              \
        __asm__ __volatile__ ↩
                                                                                  ↩
            \
        ( \
          "cpuid\n\t" ↩

              \
          "rdtsc\n\t" ↩

              \
          "mov %%edx, %0\n\t" ↩
                                                                                  ↩
              \
          "mov %%eax, %1\n\t" ↩
                                                                                  ↩
              \
          :"=r" (init_cycles_high_bits), "=r" (↩
              init_cycles_low_bits)                               \
          : /* no input operands */ ↩
                                                                          ↩
              \
          : RDTSC_CLOBBERED_REGISTERS ↩
                                                                        ↩
              \
        ); ↩

            \
        (initialization) = (((uint64_t)init_cycles_high_bits ↩
            << 32) | init_cycles_low_bits);   \
    } while(0)

#define CLOCK_COUNT_TERM(termination) ↩
                                                               \
    do { ↩

       \
        register unsigned term_cycles_high_bits, ↩
            term_cycles_low_bits;                            \
        __asm__ __volatile__ ↩
                                                                                ↩
            \
        ( \
          "rdtscp\n\t" ↩

              \
          "mov %%edx, %0\n\t" ↩
                                                                                 ↩
              \
          "mov %%eax, %1\n\t" ↩
                                                                                 ↩
            \
          "cpuid\n\t" ↩

            \
          :"=r" (term_cycles_high_bits), "=r" (↩
             term_cycles_low_bits)                          \
          : /* no input operands */ ↩
                                                                    ↩
            \
          : RDTSC_CLOBBERED_REGISTERS ↩
                                                                  ↩
            \
        ); ↩

          \
        (termination) = (((uint64_t)term_cycles_high_bits << ↩
          32) | term_cycles_low_bits);   \
    } while(0)

#define CLOCK_COUNT_TERM_PORTABLE(termination) ↩
                                                    \
    do { ↩

        \
        register unsigned term_cycles_high_bits, ↩
          term_cycles_low_bits;                       \
        __asm__ __volatile__ ↩
                                                                        ↩
          \
        ( \
          "mov %%cr0, %SUPPORTED_REG\n\t" ↩
                                                              ↩
            \
          "mov SUPPORTED_REG, %%cr0\n\t" ↩
                                                              ↩
            \
          "rdtsc\n\t" ↩

            \
          "mov %%edx, %0\n\t" ↩
                                                                        ↩
            \
          "mov %%eax, %1\n\t" ↩
                                                                        ↩
            \
          "cpuid\n\t" ↩

            \
          :"=r" (term_cycles_high_bits), "=r" (↩
             term_cycles_low_bits)                          \
          : /* no input operands */ ↩
                                                                    ↩
```

```
                \
            : RDTSC_CLOBBERED_REGISTERS ↩
↩
                \
        ); ↩
                \
        (termination) = (((uint64_t)term_cycles_high_bits << ↩
            32) | term_cycles_low_bits);    \
    } while(0)

void inline measured_loop(unsigned int i, volatile int *var)
{
    int k;

    for (k = 0; k < i; k++)
        *var = i;
}

void inline measure_exec_t(uint64_t **slots)
{
    int i, j;
    unsigned long flags;
    uint64_t initialization, termination, overhead, temp;
    volatile int variable = 0;

    CLOCK_COUNT_INIT(initialization);
    CLOCK_COUNT_INIT(temp);
    CLOCK_COUNT_TERM(temp);
    CLOCK_COUNT_TERM(termination);
    overhead = termination - initialization;

    for (i = 0; i < NO_OF_ENSEMBLES; i++)
    {
        for (j = 0; j < SAMPLES_PER_ENSEMBLE; j++)
        {
            variable = 0;

            preempt_disable();
            raw_local_irq_save(flags);

            CLOCK_COUNT_INIT(initialization);
            measured_loop(j, &variable);
            CLOCK_COUNT_TERM(termination);

            raw_local_irq_restore(flags);
            preempt_enable();

            if ( (termination - initialization) < 0)
            {
                printk(KERN_ERR "\n\n>>>>>>>>>>>>>>CRITICAL ↩
                    ERROR IN TAKING THE TIME!!!!!!\n ensemble↩
                    (%d) sample(%d) \
                         initialization = %llu, termination = ↩
                           %llu, variable = %u\n", i, j, ↩
                           initialization, termination, ↩
                           variable);
                slots[i][j] = 0;
            }
            else
                slots[i][j] = termination - initialization - ↩
                    overhead;
        }
    }
    return;
}

uint64_t var_calc(uint64_t *inputs, int size)
{
    int i;
    uint64_t acc = 0, previous = 0, temp_var = 0;

    for (i = 0; i < size; i++)
    {
        if (acc < previous)
            goto overflow;
        previous = acc;
        acc += inputs[i];
    }

    acc = acc * acc;
    if (acc < previous)
        goto overflow;
    previous = 0;

    for (i = 0; i < size; i++){
        if (temp_var < previous)
            goto overflow;
        previous = temp_var;
        temp_var+= (inputs[i]*inputs[i]);
    }

    temp_var = temp_var * size;
    if (temp_var < previous)
        goto overflow;
    temp_var =(temp_var - acc)/(((uint64_t)(size))*((uint64_t↩
       )(size)));
    return (temp_var);

overflow:
    printk(KERN_ERR "\n\n>>>>>>>>>>>>>>> CRITICAL OVERFLOW ↩
       ERROR IN var_calc!!!!!!\n\n");
    return -EINVAL;
}
```

```
static int __init module_run(void)
{
    int i = 0, j = 0, spurious = 0, k = 0;
    uint64_t **slots;
    uint64_t *variances;
    uint64_t *min_values;
    uint64_t max_dev = 0, min_time = 0, max_time = 0, ↩
       prev_min =0, tot_var=0,
             max_dev_all=0, var_of_vars=0, var_of_mins=0;

    printk(KERN_INFO "Loading CPU cycles counter module...\n"↩
       );

    slots = vmalloc(NO_OF_ENSEMBLES * sizeof(uint64_t*));

    if (!slots)
    {
        printk(KERN_ERR "unable to allocate memory for slots\↩
           n");
        return 0;
    }

    for (i = 0; i < NO_OF_ENSEMBLES; i++)
    {
        slots[i] = vmalloc(SAMPLES_PER_ENSEMBLE * sizeof(↩
           uint64_t));
        if (!slots[i])
        {
            printk(KERN_ERR "unable to allocate memory for ↩
               slots[%d]\n", i);
            for (k = 0; k < i; k++)
                vfree(slots[k]);
            vfree(slots);
            return 0;
        }
    }

    variances = vmalloc(NO_OF_ENSEMBLES * sizeof(uint64_t));
    if (!variances)
    {
        printk(KERN_ERR "unable to allocate memory for ↩
           variances\n");
        return 0;
    }

    min_values = vmalloc(NO_OF_ENSEMBLES * sizeof(uint64_t));
    if (!min_values)
    {
        printk(KERN_ERR "unable to allocate memory for ↩
           min_values\n");
        return 0;
    }

    measure_exec_t(slots);

    for (i = 0; i < NO_OF_ENSEMBLES; i++)
    {
        max_dev = 0;
        min_time = 0;
        max_time = 0;

        for (j = 0; j < SAMPLES_PER_ENSEMBLE; j++)
        {
            if ((min_time == 0) || (min_time > slots[i][j]))
                min_time = slots[i][j];
            if (max_time < slots[i][j])
                max_time = slots[i][j];
        }
        max_dev = max_time - min_time;
        min_values[i] = min_time;
        if ((prev_min != 0) && (prev_min > min_time))
            spurious++;
        if (max_dev > max_dev_all)
            max_dev_all = max_dev;

        variances[i] = var_calc(slots[i], ↩
           SAMPLES_PER_ENSEMBLE);
        tot_var += variances[i];
        printk(KERN_ERR "loop_size:%d >>>> variance(cycles): ↩
           %llu; max_deviation: %llu; min time: %llu", j, ↩
           variances[i], max_dev, min_time);
        prev_min = min_time;
    }

    var_of_vars = var_calc(variances, NO_OF_ENSEMBLES);
    var_of_mins = var_calc(min_values, NO_OF_ENSEMBLES);

    printk(KERN_ERR "\n total number of spurious min values =↩
         %d", spurious);
    printk(KERN_ERR "\n total variance = %llu", (tot_var/↩
       NO_OF_ENSEMBLES));
    printk(KERN_ERR "\n absolute max deviation = %llu", ↩
       max_dev_all);
    printk(KERN_ERR "\n variance of variances = %llu", ↩
       var_of_vars);
    printk(KERN_ERR "\n variance of minimum values = %llu", ↩
       var_of_mins);

    for (i = 0; i < NO_OF_ENSEMBLES; i++)
        vfree(slots[i]);

    vfree(slots);
    vfree(variances);
    vfree(min_values);

    return 0;
}

static void __exit module_kill(void)
{
    printk(KERN_INFO "Exiting measurement module\n");
}

module_init(module_run);
module_exit(module_kill);
```

Listing A.4: Makefile

```
# Makefile
obj-m := test-module.o

KVERSION := $(shell uname -r)
PWD := $(shell pwd)
KERNELDIR ?= /lib/modules/$(KVERSION)/build

default:
  make -C $(KERNELDIR) M=$(PWD) modules

clean:
  make -C $(KERNELDIR) M=$(PWD) clean

test: default
  # sudo sysctl kernel.nmi_watchdog=0
  sudo insmod ./test-module.ko
  lsmod | grep test-module > /home/christos/mod.txt
  sudo rmmod test-module.ko
  dmesg | tail
```

Bibliography

[1] Benchmarks: Measuring GP (GPU/APU) Cache and Memory Latencies (updated). http://www.sisoftware.net/?d=qaf=gpu_mem_latency. [Online; accessed 01-August-2013].

[2] http://linux-ima.sourceforge.net/. [Online; accessed 23-July-2013].

[3] Avoiding and identifying false sharing among threads. http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/, . [Online; accessed 20-July-2013].

[4] Align data structures on cache boundaries. http://software.intel.com/en-us/articles/align-data-structures-on-cache-boundaries/, . [Online; accessed 20-July-2013].

[5] http://www.centos.org/docs/5/pdf/Virtualization.pdf. [Online; accessed 24-July-2013].

[6] Multicore programming guide - multicore programming and applications/dsp systems. Technical report, Texas Instruments, Aug. 2012.

[7] hUMA; AMD's Heterogeneous Unified Memory Architecture Revealed. http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/60938-huma-amd-s-heterogeneous-unified-memory-architecture-revealed.html, Apr. 2013. [Online; accessed 08-August-2013].

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, Nov. 2005. ACM.

[9] S. Akhter and J. Roberts. *Multi-Core Programming - Increasing Performance through Software Multi-threading*. Intel Press, 2006.

[10] A. Baliga, V. Ganapathy, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007)*, pages 246–251, Piscataway, NJ, USA, May 2007. IEEE Press.

[11] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 77–86, Washington, DC, USA, 2008. IEEE Computer Society.

[12] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. pages 670–684, Sept. 2011.

[13] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–62, May 2006.

[14] B. Bonet and H. Geffner. Planning under partial observability by classical replanning: theory and experiments. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 1936–1941. AAAI Press, 2011.

[15] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Control flow graphs as malware signatures. In *International Workshop on the Theory of Computer Viruses*, Nancy, France, 2007.

[16] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Control flow to detect malware. In *Inter-Regional Workshop on Rigorous System Development and Analysis 2007*, 2007.

[17] D. Bovet and M. Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[18] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of the Third international conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'06, pages 129–143, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, Mar. 2007.

[20] R. Budruk, D. Anderson, and E. Solari. *PCI Express System Architecture*. Pearson Education, 2003.

[21] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. ASPLOS VI, pages 252–262, New York, NY, USA, 1994. ACM.

[22] E. Carrera and H. Flake. Automated structural classification of malware. https://www.sourceconference.com/publications/bos08pubs/carrera-AutomatedStructuralMalwareClassification.pdf, 2011. [Online; accessed 7-July-2013].

[23] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 1(1):50–60, Feb. 2004.

[24] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] S. Cesare and Y. Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing*, volume 107 of *AusPDC '10*, pages 61–70, Darlinghurst, Australia, 2010. Australian Computer Society, Inc.

[26] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. Malware normalization. Technical report, 2005.

[27] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations, 1997.

[28] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, Inc., 3rd edition, 2005.

[29] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22:378–415, Mar. 2000.

[30] U. Drepper. The cost of virtualization. *Queue*, 6(1):28–35, Jan. 2008.

[31] V. K. Garg, Ph.D. *Elements of distributed computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[32] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[33] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitive Approach.* Morgan Kaufmann Publishers (Elsevier), 5th ed edition, 2012.

[34] E. Holk. A look at gpu memory transfer. http://blog.theincredibleholk.org/blog/2012/11/29/a-look-at-gpu-memory-transfer/. [Online; accessed 01-August-2013].

[35] R. J. Hovland. Latency and bandwidth impact on gpu-systems. Master's thesis, Norweigian University of Science and Technology, Dec. 2008.

[36] J. Jones. Caches. https://www.scss.tcd.ie/Jeremy.Jones/CS3021/5%20caches.pdf, 2012. [Online; accessed 18-July-2013].

[37] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[38] V. Khera. Factors affecting false sharing on page-granularity cache-coherent shared-memory multiprocessors. Technical report, Durham, NC, USA, 1994.

[39] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[40] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference, 2004*, pages 91–100, Dec. 2004.

[41] G. Kumar. Considerations in software design for multi-core multiprocessor architectures. http://www.ibm.com/developerworks/aix/library/au-aix-multicore-multiprocessor/. [Online; accessed 20-July-2013].

[42] Y.-Y. Liu, J.-J. Slotine, and A.-L. Barabasi. Observability of complex systems. *Proceedings of the National Academy of Sciences*, 110(7):2460–2465, Jan. 2013. doi: 10.1073/pnas.1215508110.

[43] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 21–29, New York, NY, USA, 2007. ACM.

[44] R. Love. *Linux Kernel Development.* Addison-Wesley Professional, 3rd edition, 2010.

[45] T. R. McEvoy and S. D. Wolthusen. Using observations of invariant behavior to detect malicious agency in distributed environments. In *Proceedings of the Fourth International Conference on IT Incident Management and IT Forensics*, pages 55–72, 2008.

[46] T. R. McEvoy and S. D. Wolthusen. Host-based security sensor integrity in multiprocessing environments. In *Information Security, Practice and Experience: 6th International Conference, ISPEC 2010*, pages 138–152. Springer-Verlag, 2010.

[47] T. R. McEvoy and S. D. Wolthusen. An algebra for the detection and prediction of malicious activity in concurrent systems. In *Proceedings of the 2010 Fifth International Conference on Systems (ICONS 2010)*, pages 125–133. IEEE Computer Society Press, 2010.

[48] N. Mittal and V. K. Garg. On detecting global predicates in distributed computations. In *ICDCS*, pages 3–10, 2001.

[49] J. Molina and W. Arbaugh. Using independent auditors as intrusion detection systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, ICICS '02, pages 291–302, London, UK, 2002. Springer-Verlag.

[50] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[51] D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. HP Professional Series. Prentice Hall PTR, 2002.

[52] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference. ACSAC 2007.*, pages 421–430, Miami Beach, FL, 2007.

[53] T. Müller, F. C. Freiling, and A. Dewald. TRESOR runs encryption securely outside ram. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[54] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. Technical report, Sept. 2010.

[55] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[56] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: THE HARDWARE / SOFTWARE INTERFACE.* Morgan Kaufmann Publishers (Elsevier), 4th ed edition, 2009.

[57] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. *J. Syst. Archit.*, 54(8):816–828, Aug. 2008.

[58] D. Perez-Botero, J. Szefer, and R. B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[59] N. Petroni and C. P. C. S. University of Maryland. *Property-based Integrity Monitoring of Operating System Kernels*. PhD thesis, 2008. URL http://books.google.co.uk/books?id=45-RAbjz484C.

[60] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 103–115, Oct. 2007.

[61] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium*, volume 13, pages 179–194, San Diego, CA, USA, Aug. 2004. USENIX Association.

[62] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th USENIX Security Symposium*, volume 15, pages 289–304, 2006.

[63] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, Mar. 2000.

[64] J. Reinders. *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Engineer to Engineer Series. Intel Press, 2005.

[65] R. Riedmüller, M. M. Seeger, S. D. Wolthusen, H. Baier, and C. Busch. Constraints on autonomous use of standard gpu components for asynchronous observations and intrusion detection. In *Proc. 2010 2nd International Workshop on Security and Communication Networks (IWSCN)*. IEEE Computer Society Press, 2010.

[66] N. Ruest and D. Ruest. *Virtualization, A Beginner's Guide*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2009.

[67] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 3rd edition, 2004.

[68] J. Rutkowska. Beyond the CPU: Defeating Hardware Based RAM Acquisition (part I: AMD case). http://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf, Feb. 2007. [Online; accessed 25-February-2013].

[69] J. Rutkowska. Bluepilling the Xen Hypervisor. Presented at Black Hat: http://www.invisiblethingslab.com/resources/bh08/part3.pdf, 2008. [Online; accessed 6-August-2013].

[70] J. Rutkowska and R. Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. Presented at Black Hat: http://www.invisiblethingslab.com/resources/bh08/part2.pdf, 2008. [Online; accessed 6-August-2013].

[71] C. B. Sears. The elements of cache programming style. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, ALS'00, Berkeley, CA, USA, 2000. USENIX Association.

[72] M. M. Seeger. Using control-flow techniques in a security context: A survey on common prototypes and their common weakness. In *Network Computing and Information Security (NCIS), 2011 International Conference on*, volume 2, pages 133–137, May 2011.

[73] M. M. Seeger and S. D. Wolthusen. Observation mechanism and cost model for tightly coupled asymmetric concurrency. In *ICONS 2010: Proceedings of The Fifth International Conference on Systems*, volume 0, pages 158–163, Los Alamitos, CA, USA: IEEE Computer Security, 2010. IEEE Computer Society Press.

[74] M. M. Seeger and S. D. Wolthusen. Towards concurrent data sampling using gpu coprocessing. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 557–563, Aug. 2012.

[75] M. M. Seeger and S. D. Wolthusen. A model for partially asynchronous observation of malicious behavior. In *Information Security for South Africa (ISSA), 2012*, pages 1–7, Aug. 2012.

[76] M. M. Seeger, S. D. Wolthusen, C. Busch, and H. Baier. The cost of observation for intrusion detection: Performance impact of concurrent host observation. In *Information Security for South Africa 2012 (ISSA)*, pages 1–8, Aug. 2010.

[77] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, Incorporated, 2012.

[78] S. Sparks and J. Butler. Shadow walker: Raising the bar for windows rootkit detection. *Phrack*, 11(63), Aug. 2005.

[79] A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *NDSS*. The Internet Society, 2011.

[80] R. W. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley Professional, 3rd edition, 2013.

[81] P. Szor. *The Art of Computer Virus Research and Defence*. Addison Wesley Symantec Press, 1st edition, 2005.

[82] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[83] ubra. Process hiding & the linux scheduler. *Phrack*, 11(63), Jan. 2005.

[84] J. Wei, B. D. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 97–107, Washington, DC, USA, 2008. IEEE Computer Society.

[85] R. Wojtczuk. In *Black Hat*. [Online; accessed 6-August-2013].

[86] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *ACSAC*, pages 411–420. IEEE Computer Society, 2007.

[87] X. Zhang, L. Van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 239–242, New York, NY, USA, 2002. ACM.