6-5-2014

Integrity-Based Kernel Malware Detection

Feng Zhu
fzhu001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

INTEGRITY-BASED KERNEL MALWARE DETECTION

A dissertation submitted in partial fulfillment of

the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Feng Zhu

2014

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Feng Zhu, and entitled Integrity-Based Kernel Malware Detection, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Bogdan Carbunar

Xudong He

Gang Quan

Geoffrey Smith

Jinpeng Wei, Major Professor

Date of Defense: June 5, 2014

The dissertation of Feng Zhu is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2014

DEDICATION

To my advisor, my family and my friends.

ACKNOWLEDGMENTS

Thanks, everyone!

ABSTRACT OF THE DISSERTATION

INTEGRITY-BASED KERNEL MALWARE DETECTION

by

Feng Zhu

Florida International University, 2014

Miami, Florida

Professor Jinpeng Wei, Major Professor

Kernel-level malware is one of the most dangerous threats to the security of users on the Internet, so there is an urgent need for its detection. The most popular detection approach is misuse-based detection. However, it cannot catch up with today's advanced malware that increasingly apply polymorphism and obfuscation. In this thesis, we present our integrity-based detection for kernel-level malware, which does not rely on the specific features of malware.

We have developed an integrity analysis system that can derive and monitor integrity properties for commodity operating systems kernels. In our system, we focus on two classes of integrity properties: data invariants and integrity of Kernel Queue (KQ) requests.

We adopt static analysis for data invariant detection and overcome several technical challenges: field-sensitivity, array-sensitivity, and pointer analysis. We identify data invariants that are critical to system runtime integrity from Linux kernel 2.4.32 and Windows Research Kernel (WRK) with very low false positive rate and very low false negative rate. We then develop an Invariant Monitor to guard these data invariants against real-world malware. In our experiment, we are able to use Invariant Monitor to detect ten real-world Linux rootkits and nine real-world Windows malware and one synthetic Windows malware.

We leverage static and dynamic analysis of kernel and device drivers to learn the legitimate KQ requests. Based on the learned KQ requests, we build KQguard to protect KQs. At runtime, KQguard rejects all the unknown KQ requests that cannot be validated. We apply KQguard on WRK and Linux kernel, and extensive experimental evaluation shows that KQguard is efficient (up to 5.6% overhead) and effective (capable of achieving zero false positives against representative benign workloads after appropriate training and very low false negatives against 125 real-world malware and nine synthetic attacks).

In our system, Invariant Monitor and KQguard cooperate together to protect data invariants and KQs in the target kernel. By monitoring these integrity properties, we can detect malware by its violation of these integrity properties during execution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. **Motivation**

In today's world, the Internet has become an essential part of our daily life. People enjoy various services offered on the Internet such as eBay [1], which is an example of online commercial services provided on the Internet, or Fackbook [2], a representative of online social networking services. However, with more and more users on the Internet, their security becomes a serious problem. Especially, their security is significantly threatened by malware, which is software used or programmed by attackers to execute malicious logic that can damage computer system, disable computer services, steal sensitive information or get control of computer system. For instance, malware can be created and used by criminals to steal credit card information online for money purpose.

One of the most dangerous threats to systems security today is kernel-level malware such as rootkits. Because kernel-level malware is one type of malware that runs at the same privilege level as the operating systems kernel, it can affect the OS behavior for its malicious purpose. For example, a common attack of kernel-level malware is to alter the existing code and/or data of an OS for the purpose of hiding the runtime state of the system in terms of running processes, network connections, and files. Besides, kernel-level malware can add new functionalities to the OS to carry out malicious activities such as key logging and sensitive information collection. The detection of kernel-level malware is a difficult task because kernel-level malware may be able to subvert the software (normally running in user space) that is intended to find it.

To defend against kernel-level malware, it is necessary to detect its existence and then stop the execution of its malicious logic. One popular approach is to study behaviors of malware by running it in a virtual machine or isolated sandbox. Such dynamic approach can never reason about all potential behaviors. If the malware performs differently while being analyzed, or can detect the analysis itself, then the malware has a high probability to escape detection. On the other hand, static approach, which performs code analysis to infer behaviors of malware without running it, also has difficulties: (1) the source code of malware samples is not readily available and analyzing binaries brings along intricate challenges. (2) a technique called code packing, which is a transformation of a complied program, is adopted by malware authors to obfuscate the intent of the malware and raise the difficulty of code analysis.

1.2. **Contribution**

Motivated by the above challenges, we propose our integrity-based detection for kernel-level malware in this thesis: we harden the system kernel so that malware can be detected due to the violation of integrity properties triggered by its execution. Integrity properties are attributes of the target system. For example, kernel code integrity is an important integrity property of a system and many rootkits try to subvert kernel code integrity to achieve their malicious goals such as concealing malicious running processes. Protection of kernel code integrity means preventing attackers from modifying existing code in a kernel or from executing injected code with kernel privilege, over the lifetime of the system.

Specifically, this thesis focus on two classes of integrity properties: (1) *data invariants*. These invariants include properties of global data structures and serve as specifications of data structure integrity [4]. For example, they can represent critical system integrity properties such as the immutability of the Interrupt Descriptor Table

(IDT) and the system call table. Therefore, they have been checked by state-of-the-art integrity monitors [4][5]. (2) *integrity of kernel callback queue (KQ) requests*. In modern kernels, KQs are the mechanism of choice for handling events. However, KQs can also be abused by kernel-level malware to achieve their malicious goals by submitting their KQ requests. For example, the Pushdo/Cutwail spam bot has misused one KQ called Registry Operation Notification Queue in the Windows kernel to monitor, block, or modify legitimate registry operations [6]. Hence, the integrity of KQ requests is important to system security. In this thesis, we discuss how we derive specifications of legitimate KQ requests and use them to validate all KQ request.

We have developed a system to derive and monitor these two classes of integrity properties (data invariants and integrity of KQ requests) for commodity operating systems kernels, which in turn can be used to detect or mitigate kernel-level malware. The architecture of our system is demonstrated in Figure 1.1.

Generally, in order to derive the specifications for these two integrity properties, we apply static analysis on the source code of the target kernel or perform dynamic analysis of the binary code of the target system. The specifications are then fed into the code generator to generate code that can guard the integrity properties of the target kernel as its extension. Due to the difference of these two kinds of integrity properties, we separate our system into two subsystems in real implementation: one for invariants and the other one for KQs. The invariant analysis subsystem generates the source code of Invariant Monitor, and KQ analysis subsystem produces EH-Signature (properties of legitimate KQ event handlers, which is discussed in Chapter 4) collection which is used by KQguard for validating KQ requests. Invariant Monitor and KQguard work together to protect the target kernel. Details of the two subsystems are described in Chapter 3 and 4.

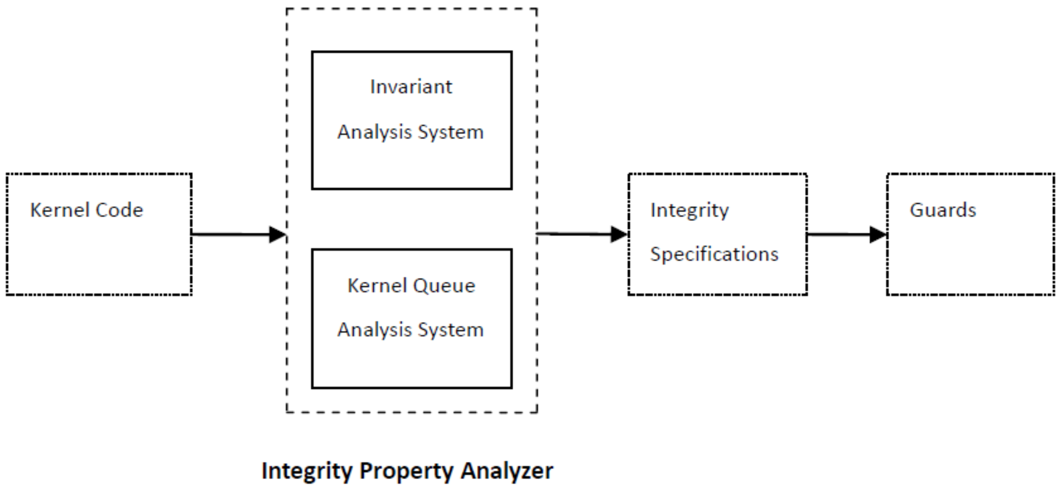

Kernel Code
Invariant Analysis System
Kernel Queue Analysis System
Integrity Specifications
Guards
Integrity Property Analyzer

Figure 1.1 The architecture of our system

1.3. **Research Goals**

The main research problems we address in this thesis are:

(1) Deriving the two classes of integrity properties---data invariants and integrity of KQ requests from a commodity operating systems kernel.

(2) Monitoring the derived integrity properties and detecting kernel-level malware that violates the monitored integrity properties.

1.4. **Outline**

The rest of this document is organized as follows. Chapter 2 reviews the modeling of data invariants and KQs as important classes of integrity properties. Chapter 3 discusses our invariant analysis for Linux kernel and WRK, and malware detection based on our Invariant Monitor. Chapter 4 describes our kernel queue analysis for Linux kernel and

WRK, and the evaluation of our KQ guard against malware. Finally, Chapter 5 summarizes the conclusions and suggests future research directions.

1.5. **Publications**

Most of the material presented in this thesis has appeared previously in the following publications:

- Feng Zhu and Jinpeng Wei (2014). "Static Analysis Based Invariant Detection for Commodity Operating Systems." Computer & Security, 43 (2014):49-63.
- Jinpeng Wei, Feng Zhu, and Yasushi Shinjo. (2011) "Static Analysis Based Invariant Detection for Commodity Operating Systems". 7th International Conference on Collaborative Computing (CollaborateCom 2011), Orlando, FL, October 15-18, 2011.
- Jinpeng Wei, Feng Zhu, and Calton Pu (2013). "KQguard: Binary-Centric Defense against Kernel Queue Injection Attacks." Computer Security–ESORICS 2013. Springer Berlin Heidelberg, 2013. 755-774.

Specifically, the first two papers are about our invariant analysis work and are the base of Chapter 3. The last paper is for our KQ analysis work and presented in Chapter 4.

2. BACKGROUND

In this chapter, we first review the basics of integrity measurement and our assumptions; then we informally define invariants in the context of integrity protection against malware attacks. Finally we introduce the concept of kernel queue(KQ) and KQ injection attack.

2.1. Background on Integrity Measurement and Security Assumptions

An integrity measurement system typically consists of three components: the target system, the measurement agent, and the decision maker [7]. Our first assumption is that the measurement agent is isolated from and independent of the target system, therefore it has a true view of the internal states (including code and data) of the target system. This is a realistic assumption due to the popularity of virtual machine monitors [8] and machine emulators such as QEMU [9], and it has also been shown that the measurement agent can run on dedicated hardware such as a PCI card [10]. Our second assumption is that measurement results are securely stored and transferred to the decision maker. This can be supported by hardware such as a Trusted Platform Module (TPM) [11]. The third assumption is that the target system's states (e.g., code and data) may be compromised by a powerful adversary who can make arbitrary modifications; therefore the decision maker can rely on very few assumptions about the trustworthiness of the target system.

Based on these assumptions, the decision maker is given a true view of the target system, and its task is to estimate the "healthiness" of the target system. The healthiness include functional correctness (e.g., a function that is supposed to reduce the priority level of a task is not modified to actually increase the priority level), and non-functional correctness (e.g., the priority level can be modified by a privileged user instead of a normal user). In the following subsections, we model the healthiness as integrity properties.

Moreover, the healthiness of the target system may change over time, because it may be under constant attacks. Therefore, the integrity of the target system may need to be periodically reevaluated.

2.2. **Data Invariant**

In this section, we first define invariants in this thesis and then discuss existing solutions for invariant analysis.

2.2.1. **Data Invariants and Their Relevance in Integrity Protection**

An invariant is a property that holds at a certain point or points in a program. In this thesis, we are interested in invariant properties that are true over all program executions. More specifically, we focus on four kinds of invariants: constant ($var == const$), membership ($var \in \{a, b, c\}$), bounds ($var \geq const$) ($var \leq const$), and non-zero ($var \neq 0$) in this thesis. The reason of this design is that other invariant analysis tools (e.g., Gibraltar [4], ReDAS [5], etc) also focus on these kinds of invariants.

Data invariants are one important class of integrity properties that concerns the expected values of program variables (other integrity properties include control flow integrity and information flow integrity). Since the behavior of a program can largely depend on its variables, manipulation of data has been an important malware attack technique. For example, many rootkits (such as Adore, Haxdoor, and SucKIT) have attacked immutable kernel data (such as system call table, System Service Descriptor Table, and SYSENTER handler function), which can be characterized as violations of constant invariants. As another example, malware can manipulate a variable used as the index into a global array, such that its value exceeds the length of the array; this out-of-bound access can trigger a kernel crash and is categorized as a violation of the bounds invariant.

On the defensive side, data invariants are the basis of many rootkit detection systems such as ReDAS [5], Copilot [10], Livewire [13], and several commercial tools (e.g., [14][15][16], and [17]). For example, the fact that a variable *var* satisfies a constant invariant (*var* == *c*) makes it easy to check its integrity: many rootkit detectors use a clean copy or hash value of a constant (e.g., *c*) as the baseline to tell whether the variable has been tampered with by a rootkit.

2.2.2. **Existing Solutions**

Several kinds of approaches have been taken to analyze a target system for invariants. Manual analysis is applicable to well-known properties such as the immutability of the Interrupt Descriptor Table (IDT) and has been used in early integrity protection systems. In response however, rootkits have moved their targets to less-known places such as device driver jump tables to evade detection, and the general trend of such attacks is towards more sophistication and stealth [18]. Eventually, manual analysis will reach a point where a human expert has difficulty understanding the logic of a system, which calls for automated tools to assist a human expert. Dynamic analysis tools such as Gibraltar [4] and ReDAS [5] can infer likely invariants of a system based on the runtime states of the target system and hypothesizing relationship among the variables at runtime. But as mentioned in Chapter 1, it is impossible for a dynamic analysis tool to cover all possible execution paths for the target system. As a result, dynamic analysis may generate false invariants. Typical solution to overcome such shortcomings is to use a large set of test cases. For example, ReDAS [5] created 70 training scenarios and 13,000 training sessions for the *ghttpd* server. However, how to systematically generate a large number of test cases that can trigger all execution paths in a program is a challenging and open research problem by itself.

2.3. **Kernel Queue**

One of the most time-critical functions of an operating system (OS) kernel is interrupt/event handling, e.g., timer interrupts. In support of asynchronous event handling, multi-threads kernels store the information necessary for handling an event as an element in a KQ, specialized for that event type. To avoid interpretation overhead, each element of a KQ contains a callback function pointer to an event handler specialized for that specific event, plus its associated execution context and input parameters. When an event happens, a kernel thread invokes the specified callback function as a subroutine to handle the event. As concrete examples, we found 20 KQs in the Windows Research Kernel (WRK) [19] and 22 in Linux kernel 2.4.32. In addition to being popular with kernel programmers, KQs also have become a very useful tool for kernel-level malware such as rootkits [6][20]. For instance, as we mentioned in Chapter 1, the Pushdo/Cutwail spam bot abuses the Registry Operation Notification Queue in the Windows kernel. This thesis includes 125 examples of real-world malware misusing KQs demonstrating these serious current exploits, and nine additional synthetic potential misuses for illustration of future dangers.

The above-mentioned kernel-level malware misuses the KQs to execute malicious logic, by inserting their own requests into the KQs. This kind of manipulation is called KQ Injection or simply KQI. Although KQI appears similar to Direct Kernel Object Manipulation (DKOM) [3] or Kernel Object Hooking (KOH) [21], it is more expressive thus powerful than the other two. While DKOM attacks only tamper with non-control data and KOH attacks only tamper with control data, KQI attacks are capable of doing both because the attacker can supply both control data (i.e., the callback function) and/or non-control data (i.e., the parameters). Moreover, KQI is stealthier than DKOM or KOH in terms of invasiveness: DKOM or KOH attacks modify legitimate kernel objects so they are invasive, while KQI attacks just insert new elements into KQs and do not have to modify any legitimate kernel objects.

Several seminal defenses have been proposed for DKOM and KOH attacks [3][4][22][23]. Unfortunately, they are not directly applicable to KQI attacks either because of their own limitations or the uniqueness of KQIs. For example, CFI [22] is a classic defense against control data attacks, but it cannot address non-control data attacks launched via KQ injection (Section 4.1.2 provides a concrete example in WRK). Gibraltar [4] infers and enforces invariant properties of kernel data structures, so it seems able to cover KQs as one type of kernel data structure. Unfortunately, Gibraltar relies on periodic snapshots of the kernel memory, which makes it possible for a transient malicious KQ request to evade detection. Petroni [24] advocates detecting DKOM by checking the integrity of kernel data structures against specifications, however, the specifications are elaborate and need to be manually written by domain experts. Finally, KQI attacks inject malicious kernel data, which makes HookSafe [23] an inadequate solution because the latter can only protect the integrity of legitimate kernel data. Therefore, new solutions are needed to defend against KQI attacks.

3. Invariant Analysis System

Remote attestation is a security mechanism that a party in a distributed environment can employ to determine whether a target computer has the appropriate hardware/software stack and configuration, so it can be trusted (i.e., it has integrity). The idea of remote attestation has been widely adopted. For example, the trusted platform modules [11] chip has become a standard component in modern computers.

Remote attestation has evolved from static attestation to runtime attestation. Traditional remote attestation techniques only ensure that a computer is bootstrapped from trusted hardware and software (e.g., operating systems and libraries), but there has been a consensus in recent years that such static attestations are inadequate [5][7]. This is because runtime attacks such as buffer overflow can invalidate the result of static attestation during the execution of the target system, so a remote challenger cannot gain high confidence in a target system even if it is statically attested [5]. In order to regain high confidence, the challenger must enhance traditional remote attestation with runtime attestation, or runtime integrity checking.

One of the determining factors of the effectiveness of runtime attestation is the attestation criteria, i.e., the expected integrity properties of the target system. Other than a few static program states (e.g., code segments and constant data), most of the runtime state of a system (normal variables, stack, and heap) cannot be trivially characterized. This uncertainty about the criteria results in two classic attestation errors: false positives and false negatives. False positives happen when the remote challenger endorses an overly stringent criterion that even an uncompromised system fails to meet; and false negatives happen when the challenger endorses an overly loose criterion that a compromised system can also meet (i.e., the remote challenger ends up trusting a corrupted computer). Obviously, both kinds of errors are undesirable for remote attestation.

The root cause for the above attestation errors is the lack of precise specifications of expected integrity properties. While *under-specification* can reduce the rate of false positives by lowering the bar for a target system, it allows a compromised system to obtain trust. On the other hand, *over-specification* errs on the side of safety to ensure that no compromised system can pass the integrity check, but it may raise too many false alarms.

In this chapter, we focus on one class of integrity properties---data invariants. As we discussed in Section 2.2.2, the existing solutions for data invariants detection are dynamic analysis and have various shortcomings. In contrast, we explore the applicability of static analysis for finding data invariants. The basic idea is to use compiler technology to analyze the behavior of a program to derive its data invariants, without actually running the program. Static analysis can overcome the limitations of dynamic analysis by exploring all execution paths. For example, if $v=v+2$ is found in the true or false branch of a conditional statement in the target program, then the property that “variable v always has a constant value at runtime” is likely false. However, a dynamic analysis tool cannot observe this assignment if the test cases do not satisfy the condition for the assignment; as a result, a dynamic analysis tool may conclude that the variable v is a constant. Since static analysis has the source code of the program, it has the advantage to reveal all conditions for assignments to a variable, so it can be more precise.

The rest of this chapter is organized as follows. Section 3.1 presents an automated invariants detection tool based on static analysis of source code. Section 3.2 discusses a thorough evaluation of our invariant detection tool by applying it to the Linux kernel and the Windows Research Kernel. Section 3.3 discusses the limitations of our approach. Section 3.4 explores related work. Section 3.5 summaries this chapter.

3.1. **Automated inference of global invariants through static analysis**

3.1.1. **Overview**

We have developed a static analysis-based system to detect invariants for commodity operating systems kernels. Figure 3.1 shows the overall architecture. We apply compiler technology to automatically analyze the control and data flows of a target kernel, e.g., assignments, function calls, and conditional statements, to support or reject hypothesis about likely invariants. For example, if a variable is assigned multiple times with different values, it is unlikely a constant.

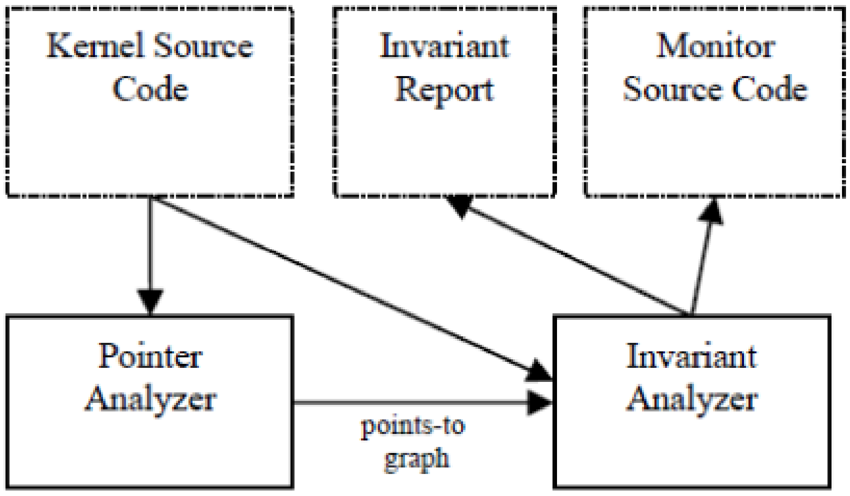


Figure 3.1 Invariant analysis architecture

Our system recognizes two kinds of assignments: direct assignment and indirect assignment. Direct assignment recognition is straightforward. Indirect assignment is mainly made through pointers in a modern kernel implemented in the C language. To recognize indirect assignment through pointers, our system has a pointer analyzer, as shown in Figure 3.1.

Our system accepts the merged kernel source code as input, which is fed into the Pointer Analyzer and the Invariant Analyzer. The Pointer Analyzer processes the kernel code and generates the points-to graph, which records the points-to set (i.e., a set of variables) of any pointer variable. The Invariant Analyzer scans the kernel source code including variable declarations, kernel functions, conditional statements, and assignment statements. When it scans an indirect assignment statement such as $*p=v$, it queries the points-to graph generated by the Pointer Analyzer and gets the set of kernel variables that can be pointed to by p, and notes that all such variables are potentially assigned once by this statement. Internally, the Invariant Analyzer has several supporting components: assignment recognition (Section 3.1.3), constant invariant recognition (Section 3.1.4), and other kinds of invariant recognition (Section 3.1.5).

The output of our system is a report of likely invariants over all global variables in the input kernel, as well as supporting evidence for manual validation (e.g., for variables that are not considered constant, our system reports the relevant assignment statement(s) that modify those variables). We have made a sample of such a report available on our web site [25]. Another output of our system is the source code for an Invariant Monitor that can be installed in the analyzed system as a kernel module to monitor the invariants detected. More details of the Invariant Monitor are presented in Section 3.2.2.1.

3.1.2. **Design Goals**

The major goal for our invariant analysis is high precision, i.e., to minimize false positive rate and false negative rate. As we discuss in the beginning of Chapter 3, both kinds of errors are undesirable for remote attestation. Below we discuss the technical challenges in both cases and our solutions when analyzing programs written in C like languages, using static detection of constant invariants ($var == const$) as an illustration.

The major reasons for false negatives are a lack of fine-granularity and imprecise pointer analysis. If the static analyzer is field-insensitive, i.e., it cannot differentiate individual fields in a C structure, it will regard an assignment to any field of a structure as an assignment to the entire structure; thus the entire structure may become a non-constant. This means that even if some fields of that structure are constant and hold critical data such as function pointers, they cannot be protected. This lack of precision obviously causes false negatives. Similarly, lack of support for array sensitivity, i.e., being unable to differentiate individual elements in an array, is another cause for false negatives. From our experience, in a modern kernel such as the Linux kernel, the majority of global data is within some structure or array, which means that a static analyzer that is field and array-insensitive is almost useless. Therefore, our invariant analyzer must be field and array-sensitive (Sections 3.1.3.1 and 3.1.3.2). Another cause of false negatives is the conservativeness of pointer analysis algorithms. If the pointer analysis algorithm is too conservative, e.g., a pointer can point to all global variables, the invariant analyzer would recognize many bogus (or impossible) assignments and thus consider a constant variable as a non-constant. Therefore, we need to develop a precise pointer analysis algorithm. We employ a precise points-to algorithm in our design (Section 3.1.3.3).

The major reason for false positives (i.e., incorrect constant invariants) is a failure to recognize legitimate assignments. This can be caused by two reasons: implicit assignments and incomplete points-to analysis. One kind of implicit assignment is assignment by assembly code: since our static analyzer does not understand assembly code, it cannot capture such assignments. Another example of implicit assignment is structure-level assignment: if variables *foo* and *bar* are defined as *struct{int a;int b}foo,bar;* then the assignment *foo = bar* implicitly modifies both *foo.a* and *foo.b*. Another cause of missing assignment recognition is related to the precision of points-to analysis: if it returns an incomplete points-to set for a pointer, the analyzer may miss legal but indirect updates to some variables through that pointer; as a result, the analyzer may mistakenly classify those variables as constants. In order to capture implicit assignments, we apply heuristics in our analyzer (Sections 3.1.3.1, 3.1.3.4, and 3.1.3.5). In order to avoid incomplete pointer analysis, we employ a precise points-to analysis algorithm (Section 3.1.3.3).

3.1.3. **Major Design Points of the Assignment Recognition**

One component of our invariant analyzer identifies assignments to variables, which is critical to the detection of constant invariants and membership invariants. For programs written in C, modifications to a variable can occur in two forms: direct assignment and indirect assignment. In the former case, the said variable is the left hand side of an assignment statement (e.g., *v* in *v=k+3*). In the latter, the said variable is assigned indirectly through a pointer that references it (e.g., *p=&v,...,*p=k+3*). In order to capture the second case, the detector needs to first find out the points-to set of the pointer (via a points-to analysis [26]), and then note that each target in the points-to set is assigned indirectly.

In the rest of this section, we discuss how our design satisfies the goals outlined in Section 3.1.2.

3.1.3.1. Field Sensitivity

To achieve the desired field sensitivity, our analyzer uses lexical names to disambiguate structure field references (e.g., *p->a* is considered a different memory location from *p->b*), in a way similar to [27]. This enables our analyzer to capture explicit assignments to structure fields. Moreover, our analyzer treats structure level assignments as implicit assignments to the individual fields. For example, if variables *foo* and *bar* are defined as *struct{int a;int b}foo, bar;* then the assignment *foo = bar;* is translated into *foo.a = bar.a; foo.b = bar.b*.

3.1.3.2. Array Sensitivity

Array sensitivity is another method for our analyzer to achieve fine-granularity. The basic idea is to treat each element of an array as an independent variable. For example, the array *int d[3]* is treated as three variables *d[0]*, *d[1]*, and *d[2]*. Our analyzer can handle arrays of arbitrary dimension. Finally, our analyzer can recognize pointers into arrays. For example, if the analyzer sees *int *p = d;* it can interpret **(p + 1)* as the same as *d[1]*.

3.1.3.3. Pointer Analysis

Our invariant analyzer performs points-to analysis in order to recognize indirect assignments through pointers. Based on the analysis in Section 3.1.2, we know that the accuracy of the points-to analysis algorithm is the key for reducing false positives and false negatives. Therefore, our pointer analyzer is based on the generalized one level flow (GOLF) algorithm [28], which is among the most precise pointer analysis algorithms, achieving precision close to Anderson’s algorithm [26]. Our pointer analyzer is built on top of the field-sensitivity (Section 3.1.3.1) and array-sensitivity (Section 3.1.3.2) capabilities to return fine-grained points-to targets. For example, it would return individual structure field *foo.b* instead of the entire structure *foo*. Finally, it can also contribute to field-sensitivity and array-sensitivity. For example, in *struct{int a;int b}bar, *p*, if *p*'s points-to set includes *bar*, then an assignment to *p->a* is considered an indirect assignment to *bar.a*.

3.1.3.4. Union Support

Our analyzer also supports unions: each field of a union is treated as an alias of other fields in the same union. This means that an explicit assignment to one field of a union is an implicit assignment to all the other fields. Therefore, if one field of a union is not a constant, other fields of the union are not constant, either.

For example, in *union uarg{int a; int b}c, c.a* and *c.b* are treated as the same variables; if *c.a* is not a constant, *c.b* is not a constant, either.

3.1.3.5. Heuristics-base Assignment Recognition

The use of assembly code in the kernel poses difficulties to our static analysis. Because our analyzer only recognizes C code, variable reads or writes by assembly code are not visible to it. One prominent example is *get_current()*, which returns a pointer to the task structure of the current process. Because this function uses assembly code, several chains of pointer dependency are broken, and our static analysis suffers inaccuracy as a result. To overcome these inaccuracies caused by assembly code, we apply a *function prototype-based heuristic*. The basic idea is to summarize the effect (in terms of assignments to the input parameters) of assembly code inside a function body to bridge the "analysis gap". For example, the function *memcpy*() copies a block of memory to another block of memory, so it can change the target memory and thus should be treated as a kind of implicit assignment. We identify this kind of functions (the functions that use assembly code) in two steps: first, our static analysis reports all functions that contain assembly code in their bodies; second, we manually analyze the reported functions to see if any assignment is performed in the assembly code. For function *get_current*() mentioned above, we assume it can return a pointer to the global variable *init_task_union.task*. In total, we apply this heuristic to 11 functions in the Linux kernel and one function in the Windows kernel.

Another difficulty to our static analysis, especially for the Windows kernel, is the use of external functions for which we do not have source code. For example, the external function *InterlockedIncrement* increments (increases by one) the value of the specified 32-bit variable as an atomic operation. To overcome this problem, we first identity this kind of functions by their *extern* tag; second, we figure out the effect of these external functions from Microsoft MSDN [29]. Then we can also apply the *function prototype-based heuristic* based on the effect of these functions. We identify eight such functions.

3.1.4. **Constant Invariant Recognition**

A second component of our invariant analyzer recognizes constant invariants, defined as having a constant value during normal execution of the system (i.e., after system initialization). Determination of constant invariants is based on the assignments to each variable, i.e., if a variable is assigned only once in its lifetime, it is a constant.

Because the definition of constant invariants only concerns the variables' value after system initialization, we treat assignments during system initialization differently than those during normal execution of the system. Specifically, a constant variable can be assigned multiple possible values during initialization, as long as it is not assigned during normal execution. On the other hand, assignments at normal execution time typically indicate that a variable is not a constant. Being assigned differently during system initialization is quite possible for some constant variables whose known-good values depend on hardware configuration; they can get several different values depending on the hardware features detected during system initialization.

In our design, each global variable is associated with a flag that indicates whether it is a constant and a legal value list that contains its possible values. Initially, all global variables are marked as constant and all legal value lists are empty.

Our invariant analyzer first scans global variable declarations and initialization functions (e.g., those with *"__init"* directives). If a global variable, which is marked as a constant invariant, is assigned a constant value, the analyzer adds this value into the variable's legal value list. On the other hand, if a global variable is assigned a non-constant value, the analyzer marks it as a non-constant.

After this scan, if a global variable's legal value list is still empty, the analyzer adds a default value into the list, based on the type of the variable (e.g., *0* for an integer variable).

Next, the constant invariant analyzer scans the remaining kernel functions. If a global variable, which is marked as a constant invariant, is assigned a non-constant value, or a constant value but the value is not in its legal value list, or a constant value in its legal value list but the length of it legal value list is greater than 1, the analyzer marks it as a non-constant (in the third case, we treat this variable as a membership invariant candidate). Note that assignments to global variables here include those indirect assignments made through pointers, i.e., when an indirect assignment is encountered, our constant invariant analyzer regards all global variables in the points-to set as being assigned.

At the end of the kernel code scanning, our analyzer generates a report about the constant invariant status of all global variables, based on their flags. For those non-constants, the report also includes the reason, e.g., the related assignment statement(s) in the kernel source code, for in-depth investigation by a human expert.

3.1.5. **Recognition of Other Kinds of Invariant**

In addition to constant invariants, our invariant analyzer can recognize several other kinds of invariants: membership invariants, bounds invariants, and non-zero invariants.

The inference of membership invariants is similar to that of constant invariants discussed in Section 3.1.4, in the sense that each variable is associated with a flag and a legal value list. The difference is that the collection of legal values happens throughout the program, not only inside the initialization functions, and the flag means whether a membership invariant can be safely reported for that variable. Specifically, when a global variable is assigned a constant value, directly or indirectly, the value is added to that variable's legal value list; however, if a non-constant value is used in the assignment, the variable is marked as unsafe for membership invariants. At the end of the analysis, membership invariants for variables still marked as safe are reported, and constants in the legal value list of each such variable are used to generate the specification of the membership invariants.

To infer bounds invariants, we apply two heuristics during the code scan: (1) if a global variable is used as an index of an array, its value should fall within the range [0, Len-1], where Len is the length of the array; (2) if a global variable is compared with a constant in a conditional statement, and one of the branches leads to a kernel crash (such as by calling *panic* or *do_exit*), then we have a constraint for this variable that its legal value should not satisfy the condition for the kernel crash. At the end of the code scan, we solve the inferred constraints for each global variable and derive the upper bound and lower bound. For example, if we have one constraint $g \geq 2$ and another constraint $g \leq 10$, the analyzer will decide a bounds invariant $2 \leq g \leq 10$ for *g*.

We recognize non-zero invariants based on a heuristic similar to the second heuristic for the bounds invariants (about a branch that causes kernel crash), where the global variable is compared against zero.

For the inferred invariants, the analysis report also includes relevant information, e.g., the conditional statement(s) in the kernel source code, which provides convenience for human validation.

We should note that the heuristics mentioned in this section are conservative in order to avoid detecting incorrect invariants, so our invariant analyzer may miss some invariants. For example, we give up the membership invariant for a variable *var* that is assigned a non-constant value (e.g., another variable *w*) because it may be very difficult to statically determine what values the other variable (e.g., *w*) may have. For another example, when a variable is compared against a non-constant in a conditional statement, we do not infer a non-zero invariant because we do not know whether the non-constant variable always contains zero or not. In this case, we may miss a non-zero invariant if the non-constant variable is always equal to zero. We can combine the result of constant invariants analysis and our existing heuristics to improve the situation. Finally, our invariant detection architecture can be extended with new heuristics once they are identified. We leave these improvements as future work.

3.1.6. **Implementation**

We implement a prototype of our static invariant detector based on the C Intermediate Language (CIL) [30]. Our pointer analyzer is implemented in 5,000 lines of Ocaml code, and our invariant analyzer is implemented in 4,300 lines of Ocaml code.

3.2. **Evaluation**

In this section, we report a large-scale evaluation of our invariant detection tool, using Linux kernel 2.4.32 and Windows Research Kernel (version Windows Server 2003) as the input kernels. Our evaluation focuses on the precision of the detected invariants and the performance of the detection tool.

3.2.1. **Metrics, Methodology, and Test Cases**

We choose two common metrics to evaluate the precision of the detected data invariants for the input kernels:

- **False positives** happen when variables whose values do not satisfy a certain invariant are mistakenly recognized as such. A monitor can generate false alarms when such variable's runtime values violate the "fake" invariants (e.g., when a variable that can legally change its value is recognized as being constant).
- **False negatives** happen when a true invariant is not recognized as such. As a result of false negatives, a monitor may fail to detect rootkits that tamper with global variables so as to violate the true invariants. For example, if a critical variable that should be constant is not monitored, a rootkit can modify it and at the same time evade detection.

We evaluate the false positive and false negative rates of the static invariant detection in two ways: (1) comparing with the result of a dynamic invariant detector, and (2) running against real software (benign or malicious). In the evaluation on the Linux kernel, the set of benign test programs includes Mozilla Firefox, the Linux Test Project [31], Iperf [32], Andrew benchmark [33], and Kernel compilation; and the malicious test programs are ten real-world rootkits such as Adore 0.42. Similarly, we evaluate WRK with a set of benign test programs: Mozilla Firefox, Internet Explorer, Notepad, WinSCP, Super PI [34], Iozone [35], 7zip [36] and Windows Driver Kit [37]. And we use nine real-world malware samples as the malicious test programs.

3.2.1.1. Comparing with a Dynamic Invariant Detector

We develop a dynamic invariant detector (as a loadable kernel module in Linux or as a Windows Device Driver in Windows) that periodically reads the values of the global variables of the kernel during a training phase and hypothesizes likely invariants satisfied by the global variables based on their observed values. For comparison purposes, this dynamic invariant detector detects the same kinds of invariants as our static invariant analyzer: constant invariants, membership invariants, bounds invariants, and non-zero invariants. To be fair in the comparison, we aim to trigger as many modifications to the global variables as possible. Therefore, we choose benign test programs (mentioned in Section 3.2.1) that represent comprehensive workloads. For example, the Linux Test Project (version 20050107) is an open source test suites that validate the reliability, robustness, and stability of Linux kernel, and it includes more than 700 test cases that test the Linux kernel in many aspects (such as system calls and file system functionality) and more than 60 test cases that exercise the basic functionalities of the network.

In the rest of this subsection, we mainly focus on the constant invariant (Section 3.2.1.2), and we briefly report the comparison on other kinds of invariants in Section 3.2.1.3.

3.2.1.2. Constant Invariants

Table 3.1 summarizes the constant invariant analysis results for the *.data* and the *.rodata* segments of Linux kernel 2.4.32, as well as the *.data* and the *.rdata* segments of WRK. The third column is the total number of global variables (with field and array-sensitivity). The fourth column shows the number of statically-detected constant invariants out of all the variables, and the fifth column shows the number of dynamically-detected constant invariants out of all the variables.

Kernel	***Segment***	***# Variables***	***# Static inv.***	***# Dynamic inv.***
Linux kernel	.data	154,132	136,778	153,978
	.rodata	4,502	4,502	4,502
WRK	.data	116,057	94,199	115,876
	.rdata	6,617	6,617	6,617

Table 3.1 Overall result of the constant invariant detection

From Table 3.1 we can see that both static and dynamic constant invariant detection achieve 100% accuracy on the *.rodata* segment and the *.rdata* segment, which is expected because variables in the *.rodata* segment and the *.rdata* segment are supposed to be constant.

The dynamic constant invariant detector reports that more than 99% of the variables in the *.data* segment of either Linux kernel or WRK are constants. Comparatively, static analysis reports 88.7% of the variables in the *.data* segment of Linux kernel and 81.2% of the variables in the *.data* segment of WRK as constants. This difference implies that despite our effort to use comprehensive test cases, they still may not be able to trigger every possible modification to global variables, so there may still exist some false constant invariants in the dynamically-detected set.

Table 3.2 gives a more in-depth comparison of the results of the static and dynamic analyzers for the *.data* segment. Since each variable can be considered constant or non-constant by each of the analyzer, there are four combinations. For example, the category "S.NC, D.C" includes all variables that are considered non-constant by the static analyzer but constant by the dynamic analyzer.

We can see that there are in total 154 variables in Linux kernel and 181 variables in WRK that both analyzers agree to be non-constant. We are confident about the correctness of the results because the dynamic analyzer classifies a variable as non-constant only if it observes that the value of the variable does change at runtime. Therefore, the non-constants reported by the dynamic analyzer must be truly non-constants.

Kernel	***Category***	***Total #***	***# Error static***	***# Error dyna.***
Linux kernel	S.NC, D.NC	154	0	0
	S.NC, D.C	17,200	18(FN)	17,182(FP)
	S.C, D.NC	0	0	0
	S.C, D.C	136,778	1(FP)	1(FP)
WRK	S.NC, D.NC	181	0	0
	S.NC, D.C	21,677	7(FN)	21,670(FP)
	S.C, D.NC	0	0	0
	S.C, D.C	94,199	1(FP)	1(FP)

Table 3.2 Comparison of the static and dynamic analysis results for the .data segment (FN: false negative; FP: false positive)

Next, we see from Table 3.2 that a large number (e.g., 17,200 in Linux kernel) of variables in the "S.NC, D.C" category are classified as non-constants by the static analyzer but constants by the dynamic analyzer. Here we cannot trust the dynamic analyzer because it may not observe legal but conditional assignments due to the incompleteness of the test coverage, and we cannot trust the static analyzer either, because its points-to analysis is conservative.

To find the ground truth about these variables, we manually verify whether they are indeed non-constants. This verification task seems daunting, but it is actually made much easier by the following facts about our static analyzer: (1) if a variable is directly modified, the assignment statement logged in the analysis report is straightforward evidence that the variable is a non-constant; (2) if a variable is only indirectly modified through a pointer, our analyzer outputs the relevant statements from the source code that support the points-to relationship, which is relatively straightforward to verify by a human (e.g., Figure 3.2 is a portion of our analysis report that shows why *ctrl_map[2]* can be indirectly modified through the pointer variable *key_map*); (3) because our analysis is array sensitive, we can generalize from one confirmed non-constant array element to all other elements in the same array. E.g., given an array *arr* of size 1024, if we confirm that *arr[0]* is a non-constant due to an assignment to *arr[i]*, then we can conclude that *arr[1]* through *arr[1023]* are all non-constants. In other words, we can confirm non-constants in batches, which significantly speed up the verification. Because of the above reasons, it took one graduate student about 20 hours to finish the verification of the 17,200 variables in the "S.NC, D.C" category of Linux kernel. As a result, we are able to confirm that 17,182 of such variables are non-constants, i.e., they can be modified by assignments at runtime. Similarly, we confirm that 21,670 out of 21,677 variables in the "S.NC, D.C" category of WRK are non-constants, which took one graduate student around 24 hours. Below we give some examples of these non-constant variables:

- The array *ctrl_map* in Linux kernel holds the values of the Control Key combinations. Figure 3.2 shows how an element of this array (e.g., *ctrl_map[2]*) can be indirectly modified through a pointer in function *do_kdsk_ioctl*. However, since our dynamic analysis test cases do not trigger function *do_kdsk_ioctl,* the dynamic invariant analyzer wrongly classifies all elements of the array *ctrl_map* as constants.

- *KeBootTime* in WRK stores the absolute time when the system was booted. One of its fields, *KeBootTime.QuadPart*, can be changed by *KeBootTime.QuadPart += TimeDelta.QuadPart* in function *KeSetSystemTime*. However, the dynamic invariant analyzer reports *KeBootTime.QuadPart* as a constant because *KeSetSystemTime* is invoked only during system boot but our test cases do not reboot the kernel.

```
<Name>ctrl_map[2]</Name>
<Invariant>No</Invariant>
<Reason1>
*(key_map + 0) = (unsigned short )(((2 << 8) | 126) ^ 61440);
vt.c:224, Indirectly modified through key_map.

Path from ctrl_map[2] to key_map:
<Label>ctrl_map[2]</Label>
<STMT>ctrl_map=&ctrl_map[2]  defkeymap.c:65</STMT>
<Label>l_473154</Label>
<STMT>key_maps[4]=ctrl_map
defkeymap.c:141</STMT>
<Label>l_479876</Label>
<STMT>key_map = key_maps[tmp.kb_table];vt.c:174
</STMT>
```

Figure 3.2 Snippet of the analysis report that shows why ctrl_map[2] can be indirectly modified through the pointer key_map

- The array *PspLoadImageNotifyRoutine* in WRK is used to hold driver-supplied callback functions that are subsequently invoked whenever an image is loaded (or mapped into memory). Specifically, it is written into by *tmp = ExCompareExchangeCallBack(& PspLoadImageNotifyRoutine[i], CallBack, (PEX_CALLBACK_ROUTINE_BLOCK)((void *)0))* in function *PsSetLoadImageNotifyRoutine*. Therefore, this array is obviously not constant. However, since our dynamic analysis test cases do not trigger any new driver-supplied callback registration, the dynamic invariant analyzer cannot see any changes to *PspLoadImageNotifyRoutine*; so it mistakenly concludes that the entire array *PspLoadImageNotifyRoutine* is constant.

We study the distribution of evidence (direct assignment and/or indirect assignment) applicable to the 17,336 confirmed non-constants in Linux kernel (including the 154 non-constants in the “S.NC, D.NC” category and the 17,182 non-constants in the “S.NC, D.C” category). We see that 13,191 non-constants (or 76% of 17,336) can be modified only indirectly through a pointer, 2,615 non-constants can be modified only directly, and the rest (1,530 non-constants) can be modified in both ways. Note that we have merged the non-constants recognized via heuristics (Section 3.1.5) into the direct or indirect group depending on whether the address of the target variable is taken. A similar study of the 21,851 confirmed non-constants in WRK (181 non-constants in the “S.NC, D.NC” category plus 21,670 non-constants in the “S.NC, D.C” category) shows that 4,321 non-constants can be modified only directly, 489 non-constants can be modified both directly and indirectly, and 17,041 non-constants (or 78% of 21,851) can be modified only indirectly through a pointer. Our observation confirms the wide-scale use of pointers in both the Linux kernel and WRK to manipulate memory and also means that for any static invariant detector of them, the points-to analysis part is critical for the overall precision.

We further look at the type and meaning of the confirmed non-constants in the “S.NC, D.C” group to see whether the classification makes sense. We coarsely divide them into several categories, such as list heads, locks, accounting information (e.g., counters), resource management information (e.g., page tables and memory zone lists), and configuration data. Table 3.3 shows example variables for each category. From this analysis, we feel that the static analysis results make sense. For example, list heads, locks, and performance counters should be dynamic, so they should not be constants. Unfortunately, our dynamic analyzer classifies this large group of non-constants in the wrong way, due to the incompleteness of test cases in our experiment. We believe that the large number of non-constants highlight the relative advantage of static analysis over dynamic analysis.

Category	***Example variables***	
	Linux kernel	***WRK***
List heads	acpi_bus_drivers.next arp_tbl.gc_timer.list.next	AcquireOpsEvent.Header.WaitListHead CmpDelayedLRUListHead
Locks	context_task_wq.lock.lock dev_base_lock.lock	AcquireOpsEvent.Header.Lock CmpHiveListHeadLock
Auditing information	kernel_module.archdata_end kernel_module.archdata_start	ExPoolCodeEnd ExPoolCodeStart
Accounting information	console_sem.count.counter con_buf_sem.count.counter	CcFastReadResourceMiss CcLostDelayedWrites
Resource mgmt data	contig_page_data.node_zonelists[0].zones[0] contig_page_data.node_zones[0].free_area[0].map	ExPoolTagTables[0] ExpNonPagedPoolDescriptor[0]
Configuration data	FDC2, FLOPPY_DMA, FLOPPY_IRQ, can_use_virtual_dma, fifo_depth	KeMaximumIncrement KeMinimumIncrement

Table 3.3 Examples of non-constants

The 18 false negatives in the "S.NC, D.C" group of Linux kernel are the fields of a global structure called *i810_fops* (e.g., *i810_fops.ioctl*, *i810_fops.read*, *i810_fops.write*, etc). Our static analyzer reports that they can be assigned indirectly through a pointer to *i810_fops*. However, our manual verification indicates that such a points-to relationship is wrong, most likely due to imprecision introduced by the unification step of the GOLF algorithm [28] that our pointer analyzer uses.

The seven false negatives in the "S.NC, D.C" group of WRK are the fields of a global structure called *KiInitialProcess* *(e.g.,* *KiInitialProcess.AddressCreationLock.Gate.Header.Absolute,* *KiInitialProcess.AddressCreationLock.Gate.Header.Type*, etc). For them, our manual analysis indicates that they should be constants, which is due to the limitation of our analyzer. The source code relevant to these false negatives is shown in Figure 3.3. First, based on the heuristics about *KeGetCurrentThread* (Section 3.1.3.5), the variable `Thread` is assigned a pointer to *KiInitialThread*; then our pointer analyzer indicates that *Thread->ThreadsProcess* is a pointer to *KiInitialProcess.* Next, since *VdmRundownDpcs* is an external function for which we do not have source code, we have to apply the *function prototype-based heuristics* (Section 3.1.3.5) for this function, which conservatively assumes that every field of the structure pointed to by the input parameter (i.e., *KiInitialProcess*) can be modified to reduce false alarms as many as possible. As a result, we have the seven false negatives.

```
In function PspExitThread:

Thread = KeGetCurrentThread();
Process = Thread->ThreadsProcess;
...
VdmRundownDpcs(Process);
```

Figure 3.3 Relevant Code Path For the False Negatives in WRK

These false negatives illustrate the limitation of our pointer analyzer, which employs a conservative points-to algorithm. However, given the total number of real constant invariants, our static analyzer still has very low false negative rate: in Linux kernel we have 141,297 real constant invariants (we have 136,778 + 4,502 = 141,280 statically detected constant invariants, 18 false negatives, and one false positive, so the total number is 141,280 + 18 - 1 = 141,297) with a false negative rate of 0.013% (18 out of 141,297); in WRK we have 100,822 real constant invariants (94,199 + 6,617 = 100,816 statically detected constant invariants, seven false negatives, and one false positive, so the total number is 100,816 + 7 - 1 = 100,822) with a false negative rate of 0.007% (7 out of 100,822).

Continue on Table 3.2, we see that no variable is classified as "S.C, D.NC", which means that no variable is classified as constants by the static analyzer but non-constants by the dynamic analyzer. Such variables, if they exist, would be false positives for the static analyzer.

Finally, for the variables in "S.C, D.C" group, both analyzers believe they should be constants. Since our static analyzer does not provide evidence for constants, we cannot verify the correctness statically. Similarly, the dynamic analyzer does not provide evidence to prove constants, either. Therefore, we decide to experimentally verify the results (Section 3.2.2).

3.2.1.3. Other Kinds of Invariants

In addition to constant invariants, our static analyzer detects 143,098 membership invariants, 41 bounds invariants and 35 non-zero invariants without false positives in Linux kernel. Our static analyzer also works well on WRK: it detects 101,897 membership invariants, 265 bounds invariants and 8 non-zero invariants from WRK with no false alarms. The size distribution of the membership invariants is shown in Table 3.4.

We can see that the detected membership invariants can have up to 45 legal values, but the majority of membership invariants have no more than two legal values. Comparatively, the dynamic analyzer can only detect less precise invariants. More specifically, the dynamic bounds detector only reports sub-ranges of statically detected bounds invariants. For example, *sel_cons* in Linux is used as an index for array *vc_cons* with a length of 64 and it can be modified by the assignment *sel_cons=fg_console* in function *set_selection*; therefore, our static detector recognizes a bounds invariant $0 \leq sel\_cons \leq 63$. However, the dynamic analyzer recognizes a tighter bounds invariant $0 \leq sel\_cons \leq 0$ because 0 is the only observed value of *sel_cons* during the training process.

Kernel	***Size of Legal Value Set***	***Number of Invariants***
Linux kernel	1	140,626
	2	2,340
	3	50
	4	49
	5	32
	11	1
WRK	1	26,129
	2	75,553
	3	192
	4	49
	5	10
	6	8
	10	2
	15	1
	45	1

Table 3.4 Size Distribution of Membership Invariants

3.2.1.4. Summary of Statically Detected Invariants

In total, our static analyzer detects 284,471 invariants in Linux kernel and 202,992 invariants in WRK. The details are shown in Table 3.5.

	Linux kernel	***WRK***
Constant Invariant	141,297	100,822
Membership Invariant	143,098	101,897
Bound Invariant	41	265
Non-zero Invariant	35	8
Total	284,471	202,992

Table 3.5 Summary of Statically Detected Invariants

3.2.2. **Experimental Evaluation**

3.2.2.1. Implementation of the Invariant Monitor

We implement an Invariant Monitor that periodically (every 30 seconds) checks the statically detected invariants in the memory of a Linux kernel 2.4.32 (the kernel that our static tool analyzed). To simply the implementation, our proof-of-concept Invariant Monitor is implemented in the form of a loadable kernel module. Other techniques such as virtual machine monitor (VMM) or dedicated hardware can also be leveraged in the implementation (e.g., Livewire [13] uses a VMM and Copilot [10] runs on a PCI card).

The monitor loops over each invariant and verifies if it is satisfied by the runtime values of the global variables. The monitor emits a warning message if any invariant is not satisfied. The list of invariants as well as their specifications is derived by the static invariant analyzer presented in Section 3.1. We also implement an Invariant Monitor for WRK as part of the kernel in a similar way.

One practical difficulty that our Invariant Monitor overcomes is the semantic gap between the monitor and the rest of the kernel – not all global variables are exposed to the monitor as symbols. For example, *msg_ctlmax* is an unresolved symbol when we try to load the monitor in Linux. For this reason and for better portability (e.g., running the monitor from a hypervisor in the future), our monitor refers to global variables by their runtime addresses rather than symbolic names. However, our static invariant analyzer reports invariants using symbolic names of global variables. Therefore, we need to bridge the gap between names and runtime addresses. To solve this problem, in Linux kernel we use the information contained in the *System.map* file, a standard file generated during the kernel compilation process, which contains a mapping from kernel variable names to their runtime addresses. In WRK, we use *wrkx86.map* file instead of *System.map* file.

A related difficulty brought by the semantic gap is the lack of offset information when our monitor makes fine-grain memory accesses to individual fields of structure variables or array elements, because *System.map* and *wrkx86.map* only provide the starting address of a structure or array variable but our monitor needs to look inside it. As a scalable approach, we leverage the power of static analysis to automatically generate code for the monitor. Specifically, the static analyzer generates code that declares pointer variables of the appropriate type, uses pointer dereferencing expressions to represent fine-grain memory accesses, and lets the compiler find out the correct offset information. For example, the static analyzer generates the code snippet in Figure 3.4 for the constant invariant *timedia_data[3].num == 8* where *timedia_data* is an array whose elements are of type *struct timedia_struct* that has a field named *num*. Here 0xc0272420 is the runtime address of the *timedia_data* array. Note that the known-good value 8 that is compared in the *if* statement is automatically supplied during the code generation because it is available to the static invariant analyzer by the time of code generation (i.e., in the *legal value list*). If the length of legal value list of a constant invariant is greater than 1 and its runtime value in the first time check is also in its legal value list, this value will be treated as the only legal value of the constant invariant in the following checks.

```
p=(struct timedia_struct*)0xc0272420;

if (((struct timedia_struct*)p)[3].num!=8)
  {printk(KERN_WARNING "Bad invariant
  timedia_data[3].num \n");};
```

Figure 3.4 One example of automatically generated code for checking invariants at runtime

3.2.2.2. Evaluation of False Positives

To estimate the false positive rate of our invariant analyzer, we run the benign test programs (Section 3.2.1) while our Invariant Monitor is running in the background. The goal is to find whether the test programs can trigger warnings of invariant violations. To maximize the detection probability of false invariants, we choose benign programs that to our knowledge exercise all important subsystems of the kernel.

We run these test programs after the kernel is fully booted and our Invariant Monitor module is running. During the long time execution of the test programs, our Invariant Monitor in Linux kernel reports warning messages about only one constant invariant, which we categorize as a false positive in the "S.C, D.C" group of Table 3.2. The associated variable of this invariant is *ipv4_devconf_dflt.rp_filter*. Our invariant analyzer misses an assignment to it mainly due to a very subtle pointer arithmetic operation by the Linux kernel. We believe that this false positive can be eliminated by modifying our static analyzer.

We perform a similar experiment in WRK with our Invariant Monitor. As a result, only warning messages for the variable *WmipDockUndockNotificationEntry* are generated. To understand why our static invariant analyzer recognizes it as a constant, we perform a manual analysis, and we find that the main reason is that the value of this variable can be changed by an external function (*IoRegisterPlugPlayNotification*) that we do not have source code. However, since we can know the effect of this function from Microsoft MSDN, we believe that this false positive can be eliminated by modifying our static invariant analyzer with heuristics based on the knowledge of such external functions.

3.2.2.3. Evaluation of False Negatives

Having false negatives means that our invariant detection does not recognize some invariants; if a rootkit manipulates the relevant variables to violate such invariants, our Invariant Monitor will not be able to detect the manipulation. One way to estimate the impact of false negatives is to run real-world rootkits on a system with our Invariant Monitor installed, and see whether our Invariant Monitor can detect them.

We select ten real-world rootkits for this purpose in Linux kernel and nine real-world malware samples in WRK. Our Invariant Monitor successfully detects all of them. We also develop a proof-of-concept malware for WRK whose target is *ExpPoolScanCount* (a bounds invariant within [0, 31]). This malware modifies the value of *ExpPoolScanCount* to 63 and causes a kernel crash when *ExpPoolScanCount* is used as an index for array *KiProcessorBlock* because 63 exceeds the length of this array. Our Invariant Monitor successfully detects the violation to the bounds invariant associated with *ExpPoolScanCount* before it can cause the kernel crash.

The overall result is shown in Table 3.6. Each rootkit (or malware sample) violates one or more constant invariants. For example, SucKIT 2 tampers with one entry of the system call table: *sys_call_table[59]*, whose known-good value should be *sys_olduname*, and Adore 0.42 tampers with 15 entries in this table. For clarity, we do not show the constant values for the invariants in Table 3.6.

Figure 3.5 shows a screenshot about how our Invariant Monitor detects Storm [38], a notorious spam botnet, in WRK. As we can see, our Invariant Monitor generates a warning message about violations to two constant invariants: *KiServiceTable[77]* whose value should be 0x8226B73A (address of *NtEnumerateValueKey)* but is changed to

0xFA92539B*; KiServiceTable[151]* whose value should be 0x826A4D28 *(*address of *NtQueryDirectoryFile)* but is changed to 0xFA9251B0. The two suspicious values (*0xFA92539B* and *0xFA9251B0*) both fall within the address range of a device driver called *kernelv.sys* that is loaded by Storm. This can help us to confirm the detection of Storm.

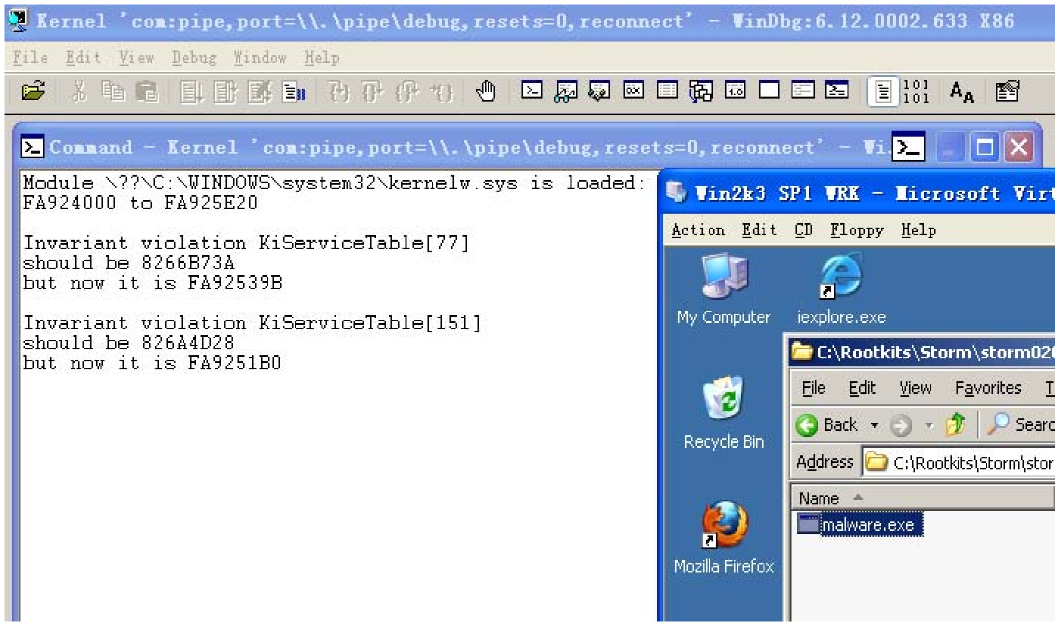
3.2.2.4. Performance Overhead

In this section, we report the performance of our prototype implementations. All experiments are performed on a server with a 2.93 GHz, 8-core Intel Xeon CPU and 16 GB of RAM.

Kernel	***Name***	***Violated Invariants***
	Real-world Rootkits	
Linux kernel	Adore 0.42	sys_call_table[2,4,5,6,18,37,39,84,106, 107,120,141,195,196,220]
	Adore 0.53	sys_call_table[1,2,6,26,37,39,120,141,220]
	All-root	sys_call_table[24]
	Kbdv2	sys_call_table[24,106]
	Kbdv3	sys_call_table[30,199]
	Modhide	sys_call_table[5]
	Phide	sys_call_table[2,37,141]
	Rial	sys_call_table[3,5,6,141,167]
	Rkit 1.01	sys_call_table[23]
	Suckit 2	sys_call_table[59]
WRK	***Real-world Malware Samples***	
	Alureon	KiServiceTable[185]
	Bot Mailer 2	IDT[0] , IDT[1] ,..., IDT[255]
	Cutwail	KiServiceTable[x], where x=68,75,77,126,256
	Haxdoor	KiServiceTable[x], where x=49,50,128,134,151,181
	Rustock.A	IDT[0] , IDT[1] ,..., IDT[255]
	Rushtock.B	IDT[0] , IDT[1] ,..., IDT[255]
	Storm	KiServiceTable[77], KiServiceTable[151]
	TDL	KiServiceTable[185]
	Trojan.Mssync	KiServiceTable[x], where x=39,43,75,77,122,125,151,181
	Proof-of-Concept Malware Sample	
	Crash Kernel	$0 \le$ ExpPoolScanCount ≤ 31

Table 3.6 Rootkit Detection Results

In the evaluation for Linux kernel, our invariant monitor and runtime analyzer run in a virtual machine with 2GB of RAM and 20GB of disk, running a Red Hat 7.3 system with Linux 2.4.32 kernel. Our static analyzer takes a merged Linux 2.4.32 kernel with 400,492 lines of C code as input, and produces the invariant report and the source code of the invariant monitor. The whole process takes 272 minutes and 4.3GB memory on average. In more detail, the pointer analyzer takes 151 minutes on average, and the invariant analyzer takes 121 minutes on average.


Kernel 'com:pipe,port=\\.\pipe\debug,resets=0,reconnect' - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command - Kernel 'com:pipe,port=\\.\pipe\debug,resets=0,reconnect' - Wi
Win2k3 SP1 WRK - Microsoft Virt
Action Edit CD Floppy Help
My Computer
iexplore.exe
Recycle Bin
Mozilla Firefox
C:\Rootkits\Storm\storm02
File Edit View Favorites
Back
Search
Address
C:\Rootkits\Storm\stor
Name
malware.exe

```
Module \??\C:\WINDOWS\system32\kernelw.sys is loaded:
FA924000 to FA925E20

Invariant violation KiServiceTable[77]
should be 8266B73A
but now it is FA92539B

Invariant violation KiServiceTable[151]
should be 826A4D28
but now it is FA9251B0
```

Figure 3.5 Detection of the invariant violation by Storm

Similarly, our invariant monitor and runtime analyzer also run in a virtual machine during the evaluation for WRK. The host operating system is Microsoft Windows XP Service Park 3 running Microsoft Virtual PC 2007 (version 6.0.156.0). The guest operating systems is Windows Server 2003 Service Pack 1 running the WRK, and it is allocated 256 MB of RAM and 20 GB of hard disk. The input to our static analyzer is a merged WRK kernel with 665,969 lines of C code. The static analysis takes 684 minutes and 6.2 GB memory on average (the pointer analyzer takes 369 minutes and the invariant analyzer takes 315 minutes).

From the above description, an astute reader may have an impression that execution time of our static analyzer is quite long. Our justification is that the static invariant detection only needs to run offline, so we have not optimized our static analyzer for speed; we leave it as future work.

We also evaluate the performance overhead of our Invariant Monitor on WRK. We first use a mircobenchmark that measures the actual time taken by the Invariant Monitor, specifically the total time spent in performing 100 invariant checks (one per 30 seconds). They consumed 20.1 seconds of CPU time in total, which happens over a time span of 3,006 seconds (50 minutes 6 seconds), so the CPU overhead of our Invariant Monitor is about 0.66%. Next, we choose five resource-intensive application benchmarks for the performance evaluation: Super PI [34], copying a directory with a total size of 1.5 GB, performing the compression and decompression of the 1.5 GB directory with 7-Zip, and downloading a 160 MB file with WinSCP. The results show that the overhead caused by our Invariant Monitor ranges from 0.41% for the file downloading to 0.62% for Super PI, which is consistent with the microbenchmark measurement.

3.3. **Limitations of Our Approach**

The static detection of data invariants presented in this thesis has some limitations. First, it requires that source code of the kernel be available, so for closed-source kernels, our tool can only be applied by the vendor of the kernel. Second, our prototype implementation of the tool cannot handle assembly code. Third, the imprecision of our pointer analyzer can introduce false positives, as is discussed in Section 3.2.1.2. Fourth, a transient attack, which modifies the value of an invariant temporarily between two checks, may bypass our Invariant Monitor due to its fixed check interval. Fifth, since our approach is based on the source code, the correctness of the derived data invariants (i.e., whether they would agree with the knowledge of a human expert) depends on the correctness of the source code. In other words, if the source code contains any error (e.g., due to coding errors or unexpected side effects), the analysis result can also be wrong according to a human expert. To overcome the first and the second limitations, we could leverage static binary analysis techniques such as Vine in BitBlaze [39] and its successor BAP [40], which provides an infrastructure for analyzing programs at the binary level. To address the third limitation, we could build a pointer analyzer based on the Anderson's algorithm [26], which has a better precision than the generalized one level flow (GOLF) algorithm. To handle the fourth limitation, we can randomize the check interval to reduce the effectiveness of a transient attack. For the last limitation, our tool is intended to assist, rather than replace, a human expert. When there is any conflict between the analysis result of our tool and the knowledge of a human expert in this area, the final judgment should be made by the human expert.

If an attacker knows about our integrity model, he might be able to write a rootkit that avoids the detection by attacking the unmonitored data. However, such an attack is out of the scope of our work. Our work is not a panacea but it still raises the bar for an attacker.

3.4. **Related Work**

In this section, we briefly discuss some related work.

Invariants detection

The Daikon invariant detector [41] is a dynamic invariant detector that generates likely invariants using program execution traces collected during sample runs. Gibraltar [4] and ReDAS [5] also adopt Daikon's idea.

Closely related to our bounds invariants detection, significant research has been performed in static detection of array bounds violations [42] or integer range analysis [43]. However, a common goal among these traditional research projects is to detect program vulnerabilities (e.g, buffer overflow), rather than derive runtime integrity properties.

Integrity measurement mechanisms

There is much previous research on integrity measurement. IMA [44] uses hashing or digital signatures to measure the software but only at load time. Recent research such as ReDAS [5] and DynIMA [45] improve IMA so that they can support runtime measurement of software integrity. Other related work includes [13], [7], [10], [24], [46], and [47]. These approaches have different mechanisms for measurement, but they do not focus on the integrity properties.

Copilot [10], a co-processor based integrity checker for the Linux kernel, checks kernel code, module code, and jump tables of kernel function pointers. Copilot did not focus on deriving integrity properties although it provided a specification language [24] later. In our thesis, we figure out the properties from analyzing the target software itself.

Livewire [13] is a host-based intrusion detection system. It uses a VMM (a modified version of VMware workstation) to monitor the states of a guest OS so that it can detect intrusions, and interposes on certain events, such as interrupts and updates to device and memory state. Like Copilot, Livewire just focuses on checking known properties but not on the identification of integrity properties.

LKIM [7] leverages the concept of contextual inspection to produce detailed records of the states of security relevant structures within the Linux kernel. However, it relies on domain knowledge to identify security relevant structures.

KOP [48] presents a nice approach for traversing the kernel memory based on extended type graph and can infer types for generic pointers with high accuracy. Although integrity checking applications can be built on top of KOP, the requirement of source code is a limitation of KOP.

Specialized integrity property measurement

There are some specialized integrity property measurements such as control flow integrity (CFI) [49] and Information flow integrity [50]. CFI, which essentially checks a sequence property of the target software, checks whether the control transfer from one function to the next is consistent with a pre-computed control flow graph. PRIMA [50] leverages the reasoning about information flows to check the integrity of a system, but it makes assumptions that there is no direct memory modification attack, e.g., information flows are triggered by well-defined interfaces (function calls or file reads). HookSafe [23] is designed to protect legitimate kernel data structures. It takes kernel hooks as input and then protects them. But some input kernel hooks are obtained manually, which implies a coverage issue. Furthermore, HookSafe only protects the hooks but not the non-control data. IndexedHooks [51] provides an alternative implementation of CFI for the FreeBSD 8.0 kernel by replacing function addresses with indexes into read-only tables, but IndexedHooks still requires source code.

Rootkits detection and recovery

There has been much research on rootkits. [10] gives us a nice survey of rootkits and detection software. A list of popular rootkits can be found from [14]. Some work such as [52] and [53] attempts to use the integrity measurement mechanisms (such as [13], [10], [46], and [47]) mentioned above to detect rootkits and recover the software from known-good copies.

Christodorescu et al [54] proposes an algorithm called semantics-aware malware detection. This static approach for malware detection uses instruction semantics to identify malicious behavior in a program from the binary of the program.

Analysis Tools

CIL [30] is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.

BitBlaze [39] focuses on binary analysis and has a static analysis component called Vine that provides an intermediate language for assembly (ILA) and an infrastructure for analyzing programs written in this language.

BAP [40] is the successor to the binary analysis techniques developed for Vine. BAP provides a formally specified intermediate language called BIL with techniques and interfaces for formally reasoning about binary programs down to the bit level.

Trusted computing

TPM (Trusted Platform Module) [11], whose state cannot be corrupted by a potentially malicious host system, is already embedded in hardware by industry vendors such as Intel. The Trusted Computing Group [55] has proposed several standards for measuring the integrity of a software system and storing the result in a TPM. Such standards and technologies have provided the root of trust for secure booting [56], and enabled remote attestation [57]. With hardware support such as TPM and application level technique such as AppCore [58], a small Trusted Computing Base can be built to facilitate integrity analysis and monitoring.

3.5. **Summary**

In this chapter, we have studied the application of static source code analysis to derive integrity properties of an operating system kernel. We design and implement automated tools that can derive data invariants out of the target kernel without running it.

To evaluate our methodology, we apply our tools to the Linux kernel 2.4.32 and the Windows Research Kernel. In Linux kernel, our tool identifies 284,471 invariants that are critical to Linux's runtime integrity. Furthermore, for the constant invariants, we compare the invariant list generated by our static analyzer with the one generated by a dynamic invariant analyzer, and find a large number of variables that can cause false alarms for the dynamic analyzer. Our tool also works for Windows Research Kernel and detects 202,992 invariants. Comparison with a dynamic invariant detector on the WRK shows similar results. Our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can potentially have very low false positive and false negative rates.

4. Kernel Queue Analysis System

Inspired by the research discussed in Section 2.3, our KQ defense endorses the general idea of using data structure invariants. However, we address the limitation of existing approaches so that our KQ integrity checking covers both persistent and transient attacks. More specifically, our defense intercepts and checks the validity of every KQ request to ensure the execution of legitimate event handlers only, by filtering out all untrusted callback requests. In [59], Wei et al develop a KQ defense for Linux (called PLCP) that employs static source code analysis to automatically derive specifications of legitimate KQ requests. However, the reliance on source code limits the practical applicability of PLCP in systems such as Windows in which there are a large number of third-party, closed source device drivers that need KQs for their normal operation.

Therefore, in this chapter, we introduce KQguard, an effective defense against KQI attacks that can support closed source device drivers. Specifically, we make the following contributions: (1) we introduce the KQguard mechanism that can distinguish attack KQ requests from legitimate KQ event handlers, (2) we employ dynamic analysis of the binary code to automatically generate specifications of legitimate KQ requests (called EH-Signatures) in closed source device drivers, (3) we build a static analysis tool that can automatically identify KQs from the source code of a given kernel, (4) we implement the KQguard in WRK and the Linux kernel, (5) our extensive evaluation of KQguard on WRK shows its effectiveness against KQ exploits (125 real-world malware samples and nine synthetic rootkits), detecting all except two of the attacks (very low false negative rate). With appropriate training, we eliminated all false alarms from KQguard for representative workloads. For resource intensive benchmarks, KQguard carries a small performance overhead of up to about 5%.

The rest of this chapter is organized as follows. Section 4.1 summarizes the problem caused by rootkits misusing KQs. Section 4.2 describes the high level design of KQguard defense by abstracting the KQ facility. Section 4.3 outlines some implementation details of KQguard for WRK, validating the design. Section 4.4 presents the results of an experimental evaluation, demonstrating the effectiveness and efficiency of KQguard. Section 4.5 outlines related work and Section 4.6 summaries this chapter.

4.1. **Problem Analysis: KQ Injection**

4.1.1. **Importance of KQ Injection Attacks**

Functionally, KQs are kernel queues that support the callback of a programmer-defined event handler, specialized for efficient handling of that particular event. For example, the soft timer queue of the Linux kernel supports scheduling of timed event-handling functions. The requester (e.g., a device driver) specifies an event time and a callback function to be executed at the specified time. When the system timer reaches the specified time, the kernel timer interrupt handler invokes the callback function stored in the soft timer request queue (Figure 4.1). More generally and regardless of the specific event semantics among the KQs, their control flow conforms to the same abstract type: for each request in the queue, a kernel thread invokes the callback function specified in the KQ request to handle the event.

KQ injection attacks (e.g., by supplying malicious callback function or data in step 1 of Figure 4.1) only uses legitimate kernel interface and it does not change legitimate kernel code or statically allocated data structures such as global variables. Therefore, syntactically a KQ injection request is indistinguishable from normal event handlers. Consider the Registry Operation Notification Queue as illustration. Using it in defense, anti-virus software event handlers can detect potential intruder malicious activity on the Windows registry. Using it in KQ injection attack, Pushdo/Cutwail [6] can monitor, block, or modify legitimate registry operations.

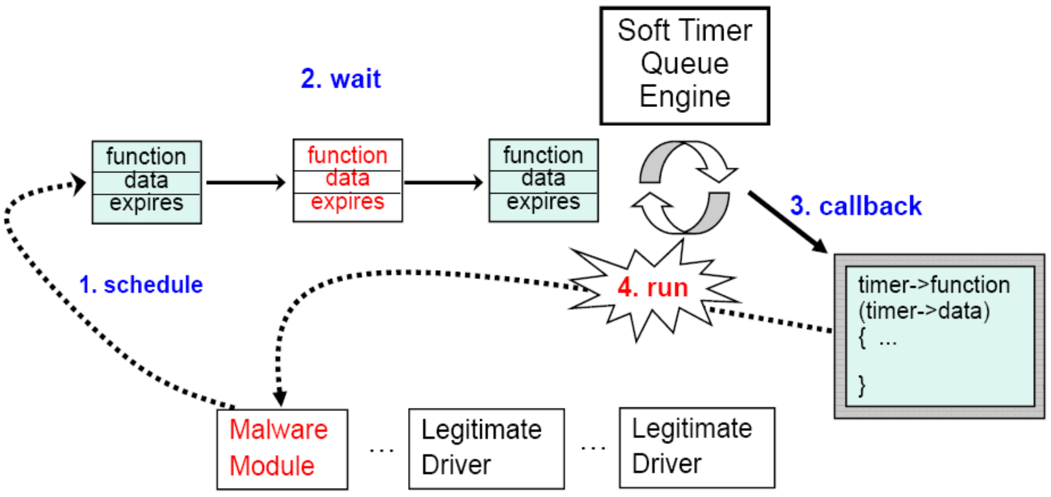


Figure 4.1 Life cycle of a timer request in Linux

Several KQ injection attacks by real world malware have been documented. First, rootkits have misused KQs to hide better against discovery. For example, the Rustock.C spam bot relies on two Windows kernel timers [60] to check whether it is being debugged/traced [73], [74] (e.g., whether KdDebuggerEnabled is true). Second, rootkits have misused normal KQ functionality for covert rootkit operations. For example, Pushdo/Cutwail botnet sets up a callback routine by invoking IoRegisterFsRegistrationChange [6], which enables it to monitor file system registrations, and Rustock.C invokes PsSetCreateProcessNotifyRoutine [61], [75] to inject code into seemingly benign processes (e.g., services.exe) so that spamming can be done in the context of an innocent process. Third, rootkits have misused KQ functionality to attack security products directly. For example, the Storm/Peacomm spam bot invokes PsSetLoadImageNotifyRoutine to register a malicious callback function that disables security products when they are loaded [76]. Table 4.2 (Section 4.4.1) shows some representative KQ hijack attacks against WRK that we have studied, which cover some of the most notorious malware today: the TDSS botnet consists of 4.5 million infected machines and is considered the most sophisticated threat [62], and Duqu [63] is believed to be closely related to the widespread Stuxnet worm [64].

4.1.2. **KQ Injection Attack Model**

The KQ injection malware listed in Table 4.2 (Section 4.4.1) misuse KQs in a straightforward way. They prepare a malicious function in kernel space and use its address as the callback function pointer in a KQ request. We call these *callback-into-malware* attacks. Since their malicious functions must be injected somewhere in the kernel space, callback-into-malware attacks can be detected by runtime kernel code integrity checkers such as SecVisor [65]. Therefore, they are considered the basic level of attack.

Unfortunately, a more sophisticated level of KQ injection attacks, called *callback-into-libc* (in analogy to return-into-libc [66], [67]), create a malicious callback request containing a legitimate callback function but malicious input parameters. When activated, the legitimate callback function may carry out *unintended* actions that are beneficial to the attacker. For example, one legitimate callback function in the asynchronous procedure call (APC) queue of the WRK is *PsExitSpecialApc*, which can cause the currently executing thread to terminate with an exit status code that is specified in the "NormalContext" parameter field of the APC request structure. Therefore, hypothetically an attacker can inject an APC request with *PsExitSpecialApc* as the callback function to force a thread to terminate with a chosen exit status code (set in the “NormalContext” field). This kind of Callback-into-libc attack can be used to shutdown an anti-virus program but make the termination appear normal to an Intrusion Detection System, by setting a misleading exit status code.

Callback-into-libc KQ injection attacks represent an interesting challenge, since they allow an attacker to execute malicious logic without injecting his own code, and the above example shows that such attacks can target non-control data (e.g., the exit status code of a thread). Therefore, they cannot be defeated by approaches that focus on control data (e.g., CFI [22]).

The design of KQguard in Section 4.2 shows how we can detect both callback-into-malware and callback-into-libc KQ injection attacks.

4.1.3. **Design Requirements of KQ Defense**

An effective KQ defense should satisfy four requirements: efficiency, effectiveness, extensibility, and inclusiveness. In this section, we outline the reasons KQguard satisfies these requirements. Some previous techniques may solve specific problems but have difficulties with satisfying all four requirements. We defer a discussion of related work to Section 4.5.

Efficiency

It is important for KQ defenses to minimize their overhead; KQguard is designed to protect KQs with low overhead, including the time-sensitive ones.

Effectiveness

KQ defenses should detect all the KQ injection attacks (zero false negatives) and make no mistakes regarding the legitimate event handlers (zero false positives); KQguard is designed to achieve this level of precision and recall by focusing on the recognition of all legitimate event handlers.

Extensibility

Due to the rapid proliferation of new devices, it is important for KQ defenses to extend their coverage to new device drivers; the KQguard design isolates the knowledge on legitimate event handlers into a table (EH-Signature collection) that is easily extensible.

Inclusiveness

A practical concern of commercial kernels is the protection of third-party, closed source device drivers; KQguard uses static analysis when source code is available and dynamic analysis to protect the closed source legitimate drivers.

4.2. **Design of KQguard**

In this section, we describe the design of KQguard as a general protection mechanism for the KQ abstract type. The concrete implementation is described in Section 4.3.

4.2.1. **Architecture Overview and Assumptions**

The main idea of KQguard is to differentiate legitimate KQ event handlers from malicious KQ injection attacks based on characteristics of *known-good* event handlers. For simplicity of discussion, we call such characteristics *Callback-Signatures*. A Callback-Signature is an effective representation of a KQ event handler (or a KQ request) for checking. One special type of Callback-Signatures is those of the legitimate KQ event handlers, and we call them *EH-Signatures*.

How to specify or discover the EH-Signatures is a practical challenge in the design of KQguard. Since legitimate KQ requests are originated from legitimate kernel or device drivers, in order to specify EH-Signatures we need to study the behavior of the core kernel and legitimate drivers. In an ideal kernel development environment, one could imagine annotating the entire kernel and all device driver code to make KQ requests explicit, e.g., by defining a KQ abstract type. Processing the KQ annotations in the complete source code will give us the exact EH-Signature collection. Unfortunately, this is not practical because many third-party closed source device drivers are unlikely to share their source code.

Therefore, our design decision is to apply dynamic binary code analysis to automate the process of obtaining a specialized EH-Signature collection that fits the configuration and usage of each system. Specifically, our design uses the architecture shown in Figure 4.2. We extend the kernel in a dedicated training environment to log (collect) EH-Signatures of KQ requests that the kernel encounters during the execution of legitimate device drivers. Then we extend the kernel in a production environment to use such learned EH-Signatures to guard against KQ injection attacks, which can be launched by malware installed in the production environment.

By employing dynamic analysis, our design does not require source code of the device drivers, thus it satisfies the inclusiveness requirement. Moreover, by having two kinds of environments, we decouple the collection and the use of EH-Signatures, which allows future legitimate drivers to be supported by KQguard: we can run the new driver in the training environment to collect its EH-Signatures and then add the new EH-Signatures into the signature collection used by the production environment. By using this method, our design satisfies the extensibility requirement.

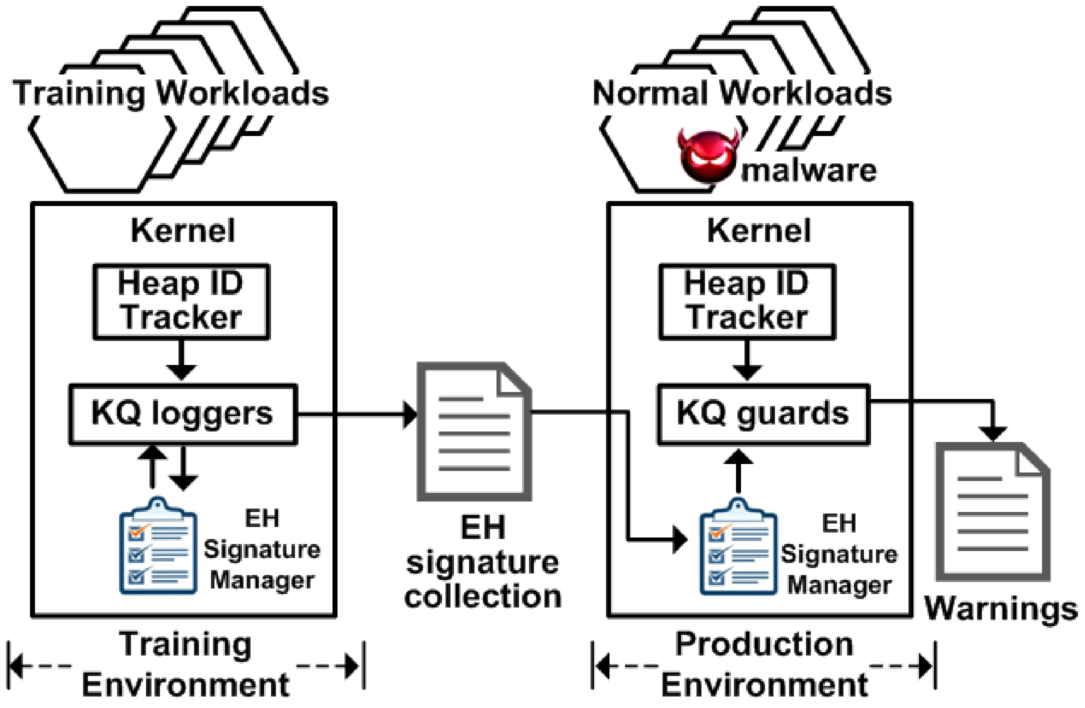


Figure 4.2 Overall Architecture of KQguard

In order to guarantee that EH-Signatures learned from the training environment is applicable to the production environment, we assume that the training environment and the production environment run the same OS and set of legitimate device drivers.

In order to guarantee that all the Callback-Signatures learned from the training environment represent legitimate KQ requests, we assume that any device driver that is run in the training environment is benign. This assumption may not hold on a consumer system because a normal user may not have the knowledge and capability to tell whether a new driver is benign or not. Therefore, we expect that KQguard is used in a strictly controlled environment (such as military and government) where a knowledgeable system administrator ensures that only benign device drivers are installed in the training environment, by applying standard security practices.

As is typical of any dynamic analysis approach, we assume that a representative and comprehensive workload is available during training to trigger all the legitimate KQ event handlers. Because some legitimate KQ requests may be made only under certain conditions, the workload must be comprehensive so that such KQ requests can be triggered and thus logged. Otherwise, KQguard may raise false alarms.

4.2.2. **Building the EH-Signature Collection**

In order to collect EH-Signatures in a training environment, we first instrument the kernel with KQ request logging capability and then run comprehensive workloads to trigger legitimate KQ requests.

4.2.2.1. Instrumentation of the Kernel to Log EH-Signatures

To collect EH-Signatures, we instrument all places in the kernel where KQ request information is available. Specifically, we extend kernel functions that initialize, insert, or dispatch KQ requests. We extend these functions with a KQ request logging utility, which generates and logs Callback-Signatures from every "raw" KQ request (i.e., with absolute addresses) submitted by the legitimate kernel and device drivers. The details of Callback-Signature generation are non-trivial and deferred to Section 4.2.5.

In general, the information contained in EH-Signatures is readily available in the kernel, although the precise location of such information may differ from kernel to kernel. It is a matter of identifying the appropriate location to instrument the kernel to extract the necessary information. Section 4.2.6 describes our non-trivial search for all the locations of these simple changes, in which we employ static source code analysis on the entire kernel. The extensions are applied to the kernel at source code level. The instrumented kernel is then rebuilt for the EH-Signature collection process.

4.2.2.2. Dynamic Profiling to Collect EH-Signatures

In this step, we run a representative set of benchmark applications using a comprehensive workload on top of the instrumented kernel. During this phase, the kernel extensions described in Section 4.2.2.1 are triggered by every KQ request.

To avoid false negatives in KQ defense, the training is performed in a clean environment to ensure no malware Callback-Signatures are included. To avoid false positives, the training workload needs to be comprehensive enough to trigger all of the legitimate KQ requests. Our evaluation (Section 4.4.4) shows a very low false positive rate, indicating the feasibility of the dynamic profiling method. In general, the issue of test coverage for large scale software without source code is a significant challenge and beyond the scope of this thesis.

4.2.3. **Validation Using EH-Signature Collection**

As shown in the “Production Environment” part of Figure 4.2, we modify the dispatcher of every identified KQ to introduce a KQ guard that checks the legitimacy of a pending KQ request before the dispatcher invokes the callback function. To perform the check, the KQ guard first builds the Callback-Signature from a pending request (detailed in Section 4.2.5), and then matches the Callback-Signature against the EH-Signature Collection. If a match is found, the dispatcher invokes the confirmed event handler. Otherwise, the dispatcher takes necessary actions against the potential threat (e.g., generating a warning message). The details of signature matching are discussed in Section 4.2.4.

To reduce performance overhead, we cache the results of KQ validation so as to avoid repeatedly checking a KQ request if its Callback-Signature has not changed since the last time it is checked. Specifically, we maintain cryptographic hashes of the “raw” KQ requests (identified by memory location) that pass the validation, so that when the same KQ request (at the recorded memory location) is to be checked again, we recalculate the cryptographic hash and compare it with the stored one. Our profiling study confirms that a significant fraction (~90%) of KQ validation is redundant because the same KQ requests are repeatedly enqueued, dispatched, dequeued, and enqueued again. Therefore, caching the validation results for such repeated KQ requests can reduce performance overhead of KQ defense.

4.2.4. **Specification of the Callback-Signatures**

A critical design issue of KQguard is the determination of the set of characteristics in the Callback-Signatures: it must precisely identify the same KQ requests in the training and production environments. On one hand, the set must not include characteristics that can vary between the two environments (e.g., the expiration time in a soft timer request) because otherwise even the same legitimate KQ requests would appear different (false positives); on the other hand, the set must include all the *invariant* characteristics between the two environments because otherwise a malicious KQ request that differs from a legitimate request only in the missing characteristics would also pass the check, resulting in false negatives. For example, the malicious KQ request in Figure 4.3.b is allowed by a KQ guard that only checks the shaded fields, although it causes a malicious function *bar_two* to be invoked; and the malicious KQ request achieves this by tampering with the “action” field of structure *se* that is not covered by the Callback-Signature. Here when the KQ request is dispatched, *foo* is invoked with *qe.data* as its parameter.

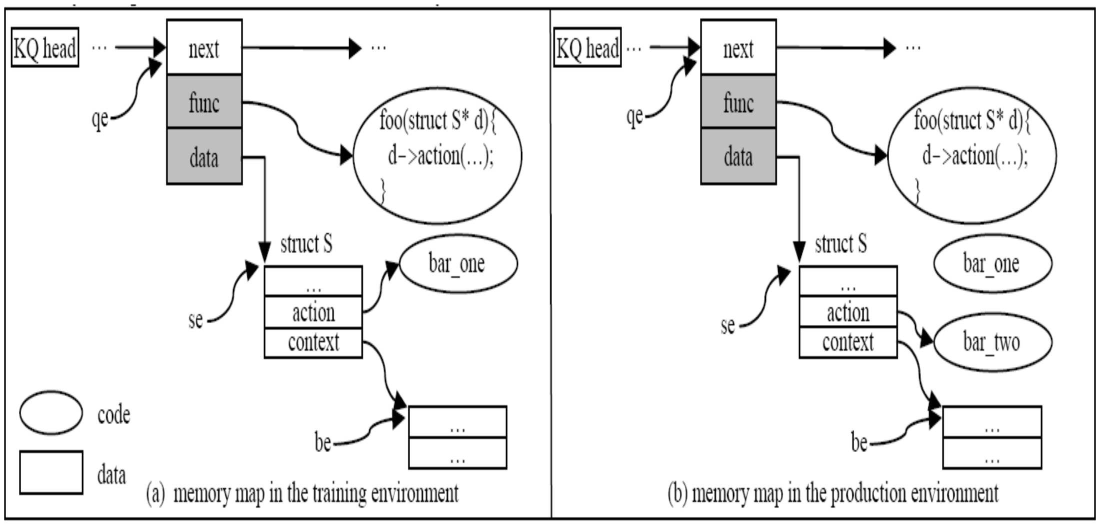


Figure 4.3 Illustration of a False Negative Caused by a Callback-Signature that Only Includes the Shaded Fields. The two KQ requests have different executions (i.e., bar_one vs bar_two), but their Callback-Signatures are the same. Here bar_two is a malicious function.

In order to minimize false negatives such as the one demonstrated in Figure 4.3, one could include more characteristics (e.g., *se.action*) into the Callback-Signatures. However, there are some challenges in doing that with closed source device drivers. Specifically, in order to realize that *se.action* is important, one can get hints from how *foo* works, but without source code, it is non-trivial to figure out that *foo* invokes *se.action*. Another possibility is to use the type information of *se* (e.g., *struct S*) to know that its "action" field is a function pointer and such information can be derived from the type of KQ request data fields (e.g., *qe.data*); unfortunately, this is often not possible because the data fields of KQ requests are often generic pointers (i.e., *void* *); in that case, one cannot figure out the type of *se* easily if it resides in a closed source device driver. Therefore, in order to support closed source device drivers, our KQ defense assumes that:

Kernel data reachable from KQ requests (e.g., *se.action*) can be identified and it has integrity in both the training and the production environments (i.e., changing of this field from *bar_one* to *bar_two* is prohibited by some other security measures).

To avoid "reinventing the wheel", we note that techniques such as KOP [48] can correctly locate kernel data such as *se.action* despite the existence of generic pointers, and techniques such as HookSafe [23] can prevent malware from tampering with invariant function pointers in legitimate kernel data structures, such as *se.action*. Moreover, both KOP and HookSafe can be used to cover even "deeper" kernel data such as *be* in Figure 4.3. Note that the inclusion of *qe.data* in the Callback-Signature is very critical because it ensures that if *qe* can pass the check performed by KQguard, *se* is a legitimate kernel data structure, and thus its "action" field can be protected by HookSafe (HookSafe is designed to protect only legitimate kernel data structures).

Note that HookSafe cannot be an alternative defense against KQ injection attacks from the top level (e.g., by ensuring that "func" and "data" fields in Figure 4.3 are not tampered with) for two reasons. First, not all top-level KQ request data structures are legitimate because malware can allocate and insert its own KQ request data structure. Second, not all top-level legitimate KQ request data structures are invariant (i.e., their values do not change) but HookSafe can only protect invariant kernel data. We have observed multiple cases in the APC queue of the WRK in which top-level legitimate KQ requests change their values during normal execution. For example, *IopfCompleteRequest* (in WRK\base\ntos\io\iomgr\iosubs.c) inserts an APC request with callback function *IopCompleteRequest* (in WRK\base\ntos\io\iomgr\internal.c); when this APC request is dispatched (i.e., *IopCompleteRequest* is invoked), its callback function field is changed to *IopUserCompletion* before it is inserted back to the APC queue.

To summarize the above discussion, (1) we need to support closed source device drivers, (2) we need a way to defend against KQ injection attacks from the top level, and (3) techniques are available to guard deeper kernel data reachable from KQ requests. Based on these three observations, in this chapter we choose a Callback-Signature format that focuses on KQ request level (the top level) characteristics: (callback_function, callback_parameters, insertion_path, allocation). Here callback_function is the callback function pointer stored in a KQ request, callback_parameters represents the relevant parameters stored in it, insertion_path represents how the KQ request is inserted (by which driver? along which code path?), and allocation represents how its memory is allocated (global, heap, or stack? by which driver?).

Each characteristic in our Callback-Signature is important for effective KQ guarding. callback_function is used to protect the kernel against callback-into-malware attacks, and both callback_function and callback_parameters are used to protect the kernel against callback-into-libc attacks (Section 4.1.2). Furthermore, insertion_path and allocation provide the context of the KQ request and thus can also be very useful. For example, if KQguard only checks callback_function and callback_parameters, malware can insert an *existing* and *legitimate* KQ request object LKQ if it can somehow benefit from the dispatching of LKQ (e.g., resetting a watchdog timer).

To ensure that the signature matching of a KQ request observed during the production use and one observed during the training can guarantee the same *execution*, we need to make sure that the code and static data of the core kernel and legitimate device drivers have integrity in the production environment. We also need to ensure that malware cannot directly attack KQ guards, including their code and the EH-Signature collection. We leverage the Xen [8] hypervisor to satisfy the above requirements. The implementation details is discussed in Section 4.3.3.

4.2.5. **Generation of Callback-Signatures from KQ Requests**

In both EH-Signature collection (Section 4.2.2) and KQ request validation (Section 4.2.3), Callback-Signatures need to be derived from raw KQ requests. This is called Callback-Signature generation and we discuss the details in this subsection.

4.2.5.1. Motivation for Delinking

As we discuss in Section 4.2.4, a Callback-Signature is a tuple (callback_function, callback_parameters, insertion_path, allocation). Since callback_function and callback_parameters correspond to fields in KQ requests (e.g., the "func" field of *qe* in Figure 4.3), it seems that we can simply copy the value of those fields into a Callback-Signature. However, when a Callback-Signature contains a memory reference (e.g., a parameter that points to a heap object), we have to overcome one challenge: namely, what the KQ loggers and the KQ guards can directly observe is an absolute memory address; however, the absolute addresses of the same variable or function can be different in the training and production environments, for example, when they are inside a device driver that is loaded at different starting addresses in the two environments. Therefore, if we use absolute addresses in the Callback-Signatures, there will not be a match for the same callback function, which results in false positives.

In order to resolve this issue, we raise the level of abstraction for memory references in the Callback Signatures so that variations at the absolute address level can be tolerated.

For example, we translate a callback function pointer (absolute address) into a unique module ID, plus the offset relative to the starting address of its containing module (usually a device driver, and we treat the core kernel as a special module). Under the assumption that the kernel maintains a uniform mapping of module location to module ID, the pair (module ID, offset) becomes an invariant representation of the callback function pointer independent of where the module is loaded. This kind of translation is called *delinking*. Due to delinking, our approach can also work with address space layout randomization: ASLR [68] randomizes the memory address space layout of a running process to make it more difficult to exploit existing vulnerabilities (e.g., buffer overflows). It works by randomly rearranging the positions of a process's code section, stack, etc in its address space. Note that ASLR can only add an offset to the starting position of the code section of a module, but it does not rearrange the internal layout of the code section. Since our Callback-Signature definition does not rely on the absolute starting address of the code section, we can support ASLR.

4.2.5.2. Details of Delinking

KQguard delinks memory references (i.e., pointers) in different ways depending on the allocation type of the target memory. As Figure 4.4 shows, there can be three types of allocations: global variable, heap variable, and local variable.

The pointer to a global variable is translated into (module ID, offset), in the same way as the callback function pointer (Section 4.2.5.1). There can be two kinds of global variables depending on whether they reside in a device driver inside the kernel or in a user-level library (e.g., a DLL on Windows). We care about user-level global variables because some KQ parameters reference user-level memory (e.g., the APC queue on

Windows). We regard device drivers and user-level libraries uniformly as modules and we modify the appropriate kernel functions to keep track of their address ranges when they are loaded (e.g., *PspCreateThread* for DLLs).

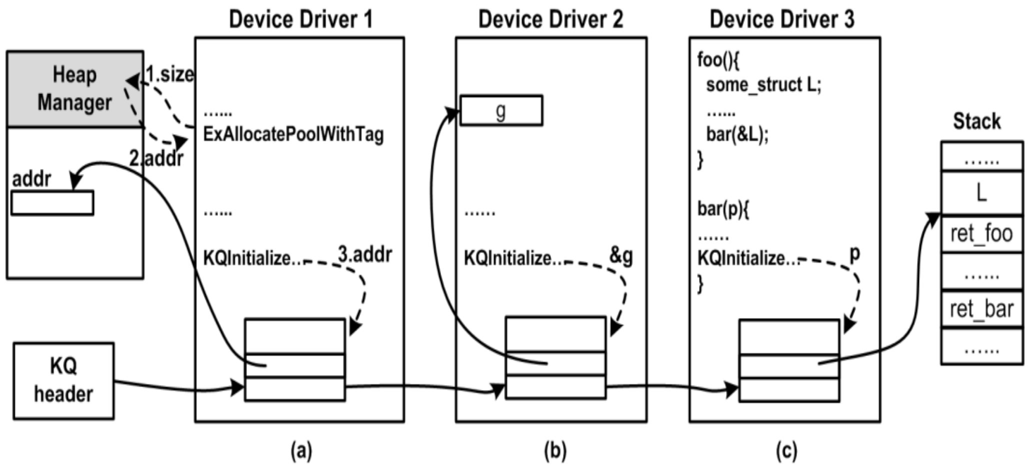


Figure 4.4 Illustration of Different Allocation Types of Pointers: (a) Heap Variable, (b) Global Variable, (c) Local Variable

The pointer to a heap object is translated into a call stack that corresponds to the code path that originates from a requester (e.g., a device driver) and ends in the allocation of the heap object. We use a call stack rather than the immediate return address because the immediate return address may be in some wrapper function for the heap allocation function and a device driver can call a function at the top of the call chain to allocate a heap object (for example, *atapi.sys* calls *IoCreateDevice* to create a heap object in Figure 4.5). Since most kernels do not maintain the request call stack for allocated heap objects, we instrument their heap allocator functions to collect such information, and the instrumentation is called Heap ID Tracker in Figure 4.2. Specifically, the Heap ID Tracker traverses the call stack frames backwards until it reaches a return address that falls within the code section of a device driver or it reaches the top of the stack; if no device driver is found during the traversal, the core kernel is used as the requester; all return addresses encountered during this traversal are part of the call stack, and each of them is translated into a (module ID, offset) pair, in the same way as the callback function pointer discussed in Section 4.2.5.1. Similar to global variables, our delinking supports two types of heap objects: kernel-level and user-level.

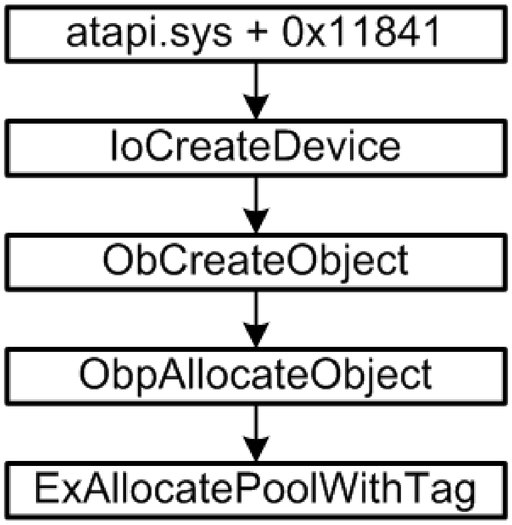


Figure 4.5 Example Heap Allocation Call Stack

The pointer to a local variable is translated into a pair (call_stack, l_offset). The call stack starts in a function where a KQ request is inspected (e.g., in a KQ insertion function), and it stops in the function that contains the local variable (e.g., *L* in function *foo* in Figure 4.4.c). Each return address encountered during the traversal is translated into a (module ID, offset) pair. Finally, l_offset is the relative position of the local variable in its containing stack frame. For example, if [ebp-8] is used to represent the local variable, l_offset is 8. We have not observed any pointers to user-level local variables, so we do not cover the translation for pointers to user-level local variables.

Because the static type of a KQ request data (e.g., the "data" field of a soft timer request structure) is often a generic pointer (i.e., *void* *), we have to detect its actual type at runtime. Given the raw value of a piece of KQ request data, we run a series of tests to decide the suitable delinking for it if it is considered a pointer. First, we test whether the raw value falls within the address range of a loaded driver or a user-level library to decide whether it should be delinked as a pointer to a global variable. If the test fails, we test whether it falls within the address range of an allocated heap object to decide whether it should be delinked as a pointer to a heap variable. If this test also fails, we test whether it falls within the address ranges of the stack frames to see whether it should be delinked as a pointer to a local variable. If this test still fails, we determine the KQ data to be a non-pointer, and no delinking is performed.

4.2.6. **Automated Detection of KQs**

Since every KQ can be exploited by malware (part of the attack surface), we need to build the EH-Signatures for all of KQs. But before we can guard a KQ, we must first know its existence. Therefore, we design and implement a KQ discovery tool that automates the process of finding KQs in a kernel by analyzing its source code. Since kernel programmers are not intentionally hiding KQs, they usually follow similar programming patterns that our tool uses effectively:

- A KQ is typically implemented as a linked list or an array. In addition to insert/delete, a KQ has a dispatcher that operates on the corresponding type.

- A KQ dispatcher usually contains a loop to act upon all or a subset of queue elements. For example, *pm_send_all* in Figure 4.6 contains the dispatcher loop for the Power Management Notification queue of Linux kernel 2.4.32.

- A KQ dispatcher usually changes the kernel control flow, e.g., invoking a callback function contained in a queue element.

```
/* linux-2.4.32/kernel/pm.c */
int pm_send_all (pm_request_t rqst, void *data)
{
  ……
  entry = pm_devs.next;
  while (entry != &pm_devs) {
    struct pm_dev *dev=list_entry(entry, struct pm_dev, entry);
    if (dev->callback) {
      int status = pm_send(dev, rqst, data);
      ……
    }
    entry = entry->next;
  }
  ……
}

int pm_send(struct pm_dev *dev, pm_request_t rqst, void *data)
{
  ……
  status = (*dev->callback)(dev, rqst, data);
  ……
}
```

Figure 4.6 Details of the Power Management Notification Queue on Linux Kernel 2.4.32

Based on the above analysis, the KQ discovery tool recognizes a KQ in several steps. It starts by detecting a loop that iterates through a candidate data structure.

Then it checks whether a queue element is derived and acted upon inside the loop. Next, our tool marks the derived queue element as a *taint* source and performs a flow-sensitive taint propagation through the rest of the loop body; this part is flow-sensitive because it propagates taint into downstream functions through parameters (e.g., *dev* passed from *pm_send_all* to *pm_send* in Figure 4.6). During the propagation, our tool checks whether any tainted function pointer is invoked (e.g., *dev->callback* in *pm_send* in Figure 4.6), and if that is the case, it reports a candidate KQ. Here we omit further details, but the results (e.g., KQs found in WRK) are interesting and discussed in Section 4.3.

4.3. **Implementations of KQguard**

In this section, we discuss our KQguard design (Section 4.2) that is implemented on the WRK and Linux kernel.

4.3.1. **KQguard Implementation on WRK**

Our implementation on the WRK is consists of about 3,900 lines of C code and 2,003 lines of Objective Caml code, which can be divided into four components:

Construction of Callback-Signatures in WRK

In order to collect the Callback-Signatures for the 20 KQs in the WRK, we instrument the kernel in two sets of functions. The first set of functions initialize, insert, or dispatch KQs and our instrumentation consists of 600 lines of C code. Some representative KQ dispatcher functions instrumented are list in Table 4.1.

To support delinking of Callback-Signatures, we instrument the device driver loader function (*IopLoadDriver*) and the thread creation function (*PspCreateThread*), and we also instrument heap allocation or deallocation functions (*ExAllocatePoolWithTag*, *ExFreePool*, *NtAllocateVirtualMemory*, and *NtFreeVirtualMemory*) to keep track of the address ranges of allocated heap memory blocks and the call stack to the heap allocation function. Our instrumentation of the heap allocator / deallocator consists of 800 lines of C code.

Automated Detection of KQs for the WRK

We implement the KQ discovery algorithm (Section 4.2.6) based on static source code analysis, using the C Intermediate Language (CIL) [69]. Our implementation consists of 2,003 lines of Objective Caml code. We applied the KQ discovery tool to the WRK source code (665,950 lines of C), 20 KQs were detected (seven of them are mentioned in Table 4.2 in Section 4.4.1 and the rest can be found in [70]), and they include all the KQs that we are aware of, which suggests the usefulness of our KQ discovery algorithm. However, whether these 20 KQs cover *all* KQs in the WRK is an interesting and open question.

KQ Name	***Queue header variable***	***#param tainted***	***Name of Dispatcher Function(s)***
Windows Research Kernel			
I/O timer queue	IopTimerQueueHead	2	IopTimerDispatch
RegistryCallback queue	CmpCallBackVector	1	CmpCallCallBacks
Load image notification queue	PspLoadImageNotifyRoutine	0	PsCallImageNotifyRoutines
APC queue	Part of a thread structure	3	KiDeliverApc
Process creation/deletion notification queue	PspCreateProcessNotifyRoutine	0	PspCreateThread, PspExitProcess
File system registration change notification queue	IopFsNotifyChangeQueueHead	0	IoRegisterFileSystem, IoUnregisterFileSystem, IoRegisterFsRegistrationChange
Callback object queue	ExpInitializeCallback	1	ExNotifyCallback
Linux Kernel			
Tasklet queue	tasklet_vec[], tasklet_hi_vec[]	1	tasklet_action, tasklet_hi_action
Packet type queue	ptype_all, ptype_base[]	1	dev_queue_xmit_nit, netif_receive_skb
Power management notification queue	pm_devs	1	pm_send_all
INET protocol handlers queue	inet_protos[]	0	ip_local_deliver_finish, icmp_unreach

Table 4.1 Representative Automatically Detected KQs

Callback-Signature Collection Management

We developed a set of utility functions to manage the Callback-Signatures, including the EH-Signatures. These functions support the generation, comparison, insertion, and search of Callback-Signatures. They are implemented in 2,200 lines of C code.

Validation of Callback-Signature in WRK

We instrument the dispatcher of every identified KQ in the WRK in the production environment so that the dispatcher checks the legitimacy of a pending KQ request before invoking the callback function (Section 4.2.3). Our instrumentation consists of about 300 lines of C code.

4.3.2. **KQguard Implementation in Linux kernel**

Our Linux implementation of KQguard follows the same conceptual design as our WRK implementation in Section 4.3.1. Of course, the names and syntax of specific functions that we instrument are different. For example, the heap object identification instruments kmalloc and kfree, which are specific to Linux.

We apply the KQ detector to Linux kernel and it found 22 KQs. Four representative KQs are listed in Table 4.1. Perhaps not surprisingly, we found differences as well as similarities between WRK and Linux. As an example of differences, WRK KQs are implemented both as linked lists and arrays, but all Linux KQs are implemented as linked lists.

To validate the pending KQ requests in Linux, we instrument the dispatcher functions for the 22 KQs detected. The retrieval of Callback-Signature follows the same process as the WRK implementation, and the EH-Signature management functions were ported from our WRK implementation.

4.3.3. **KQguard Protection**

Specifically, we run the guest kernel (with KQ guard) on top of Xen 4.1.2 hypervisor and extend the shadow-based memory management of Xen to write-protect the code and static data of the guest kernel. Note that this protection covers KQ guards and the EH-signature collection because they are part of the guest kernel. For this purpose, we add a new hypercall to Xen. This hypercall allows the guest kernel to request memory regions (whose size can be any number of bytes) in its address space to be write-protected. If a page contains any portion of a protected region, the hypervisor sets the protection bit of the page table entry of that page to read-only in the shadow page table. At runtime, the guest kernel utilizes this hypercall to protect the memory regions that contain code and static data (of the guest kernel). Since the hypercall accepts two physical addresses as its input parameters which are the start (physical) address and the end (physical) address of the memory region to be protected, we translate the virtual addresses (of the start and the end of a memory region) to physical addresses (for the reason that we can directly get the virtual addresses of a memory region but not its physical addresses) and then invoke the hypercall. We also modify the page fault handler of Xen so that any write access to the protected regions on this kind of page will be denied, but legitimate writes to other parts of this kind of page is able to go through as normal.

The instrumentation (of the hypercall mentioned above and relevant utility functions) on Xen hypervisor is consists of 1,000 lines of C code.

4.4. **Evaluation of KQguard**

In this section, we report the evaluation results of the WRK implementation of KQguard. We evaluate both the effectiveness and efficiency of KQguard through measurements on production kernels. By effectiveness we mean precision (whether it misidentifies the attacks found, measured in false positives) and recall (whether it misses a real attack, measured in false negatives) of KQguard when identifying KQ injection attacks. By efficiency we mean the overhead introduced by KQguard. In both the training and the production systems used in our evaluation, the hardware is a 2.4 GHz Intel Xeon 8-Core server with 16 GB of RAM, and the operating system is Windows Server 2003 Service Pack 1 running the WRK.

4.4.1. **Real-World KQ Injection Attacks**

We start our evaluation of KQguard effectiveness by testing our WRK implementation (Section 4.3) against real-world KQ injection attacks in Windows OS. Since malware technology keeps advancing, we focus on the most recent and the most influential malware samples that represent the state of the art. Specifically, we chose 125 malware samples from the top 20 malware families [71] and the top 10 botnet families [72]. These samples are known to have KQ injection behaviors.

Overall, our test confirmed that 98 samples inject the APC queue, 34 samples inject the DPC queue, 32 samples inject the load image notification queue, 20 samples inject the process creation/deletion notification queue, four samples inject the file system registration change queue, four samples inject the registry operation notification queue, and two samples inject the system worker thread queue.

Table 4.2 reports the results of 10 representative spam bot samples. We started with malware with reported KQ injection attacks, which are marked with a "√" with citation. We were able to confirm some of these attacks, shaded in gray. The rows with shaded "√" without citations are confirmed new KQ injection attacks that have not been reported by other sources. For example, Rustock.J injects an APC request with a callback function at address *0xF83FE316*, which falls within the address range of a device driver called msliksurserv.sys that is loaded by Rustock.J; this APC request raises an alarm because it does not match any of the EH-Signatures we have collected.

For all the malware that we were able to activate (the Rustock.C sample failed to run in our test environment), we confirmed the reported KQ injection attacks, except for the Duqu attack on load image notification queue and Storm on the APC queue.

KQ / Malware	Timer/ DPC	Worker Thread	Load Image	Create Process	APC	FsRegistrationChange	RegistryOp Callback
Rustock.C	√ [73][74]			√ [75]	√ [75]		
Rustock.J			√	√	√		
Pushdo	√			√ [6]	√	√ [6]	√ [6]
Storm	√		√ [76]		√ [77]		
Srizbi	√				√		
TDSS			√		√	√	
Duqu	√		√ [63]		√		
ZeroAccess	√	√ [78]			√ [78]		√
Koutodoor	√			√			
Pandex					√		
Mebroot	√						

Table 4.2 Known KQ Injection Attacks in Representative Malware

The Rustock samples show that malware designers have significant ability and flexibility in injecting different KQs. Concretely, Rustock.J has stopped using the timer queue, which Rustock.C uses, but Rustock.J started to use the load image notification queue, which Rustock.C does not. This may have happened to Duqu's attack on the same queue, or Duqu does not activate the attack on load image notification queue during our experiment. Overall, our evaluation indicates that KQguard can have a low false negative rate because it detects all except two of the KQ injection attacks by 125 real-world malware samples.

4.4.2. **Protection of All KQs**

In addition to real world malware, we create synthetic KQ injection attacks for two reasons. First, nine KQs have maximum queue length of zero during the testing in Section 4.4.1, suggesting that malware is not actively targeting them for the moment; however, the Rustock evolution shows that malware writers may consider such KQs in the near future, so we should ensure that guards for such KQs work properly. Second, the malware analyzed in Section 4.4.1 belongs to the callback-into-malware category. Although there have been no reports of callback-into-libc attacks in the wild, it is important to evaluate the effectiveness of KQ-guard for both kinds of attacks. Therefore, for completeness, we developed test Windows device drivers for each of the KQs that have not been called and we have confirmed that our KQ defense can detect all the test drivers, which suggests that our defense is effective against potential and future KQ injection attacks.

4.4.3. **Protection of KQguard**

As we mentioned in Section 4.3.3, the code of KQguard and the EH-Signature collection are under write-protection. To verify the effectiveness of the write-protection, we create three synthetic attacks that try to alter the code of KQguard and five synthetic attacks that try to modify the data of EH-Signature collection. We launch these attacks after

KQguard code the EH-Signature collection are write-protected. As a result, all eight attacks are all detected and prevented.

4.4.4. **False Alarms**

We have experimentally confirmed that it is possible to reduce the false positives of KQ guarding to zero. This is achievable when the training workload is comprehensive enough to produce the full EH-Signature collection.

We first collect EH-Signatures on a training machine with Internet access. We repeatedly log in, run a set of normal workload programs, and log off. In order to trigger all possible code paths that insert KQ requests, we actively do the above for fifteen hours. During this process, we gradually collect more and more EH-Signatures until the set does not grow. At the end of training, we collect 813 EH-Signatures. The set of workload programs include Notepad, Windows Explorer, WinSCP, Internet Explorer, 7-Zip, WordPad, IDA, OllyDbg, CFF Explorer, Sandboxie, and Python.

Next we feed the collected EH-Signatures into a production machine with KQ guarding and use that machine for normal workloads as well as the KQ injection malware evaluation and the performance overhead tests. During such uses, we observe zero false alarms. The normal workload programs include the ones mentioned above as well as others such as Firefox not used in training.

While the experimental result appears encouraging, we avoid making a claim that dynamic analysis can always achieve zero false positives. For example, the APC queue has 733 EH-Signatures, such EH-Signatures have 14 unique callback functions (Figure 4.7), and the most popular callback function is *IopCompleteRequest*, occurring in 603 EH-Signatures. While these 603 EH-Signatures share the same callback function, their insertion paths originate from 51 device drivers, two DLLs, and the core kernel, so the average number of EH-Signatures per requester (e.g., a device driver) is 11, and the largest number is 45 (from the driver ntfs.sys). This result implies that there can be potentially many code paths within a driver that can prepare and insert an APC request with the same callback function, which may or may not be triggered in our training.

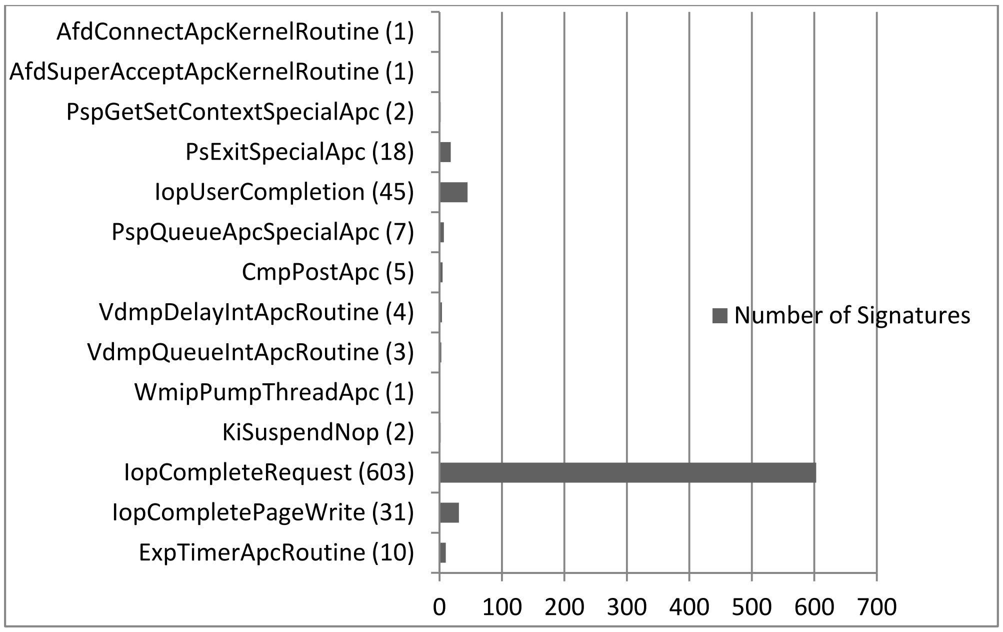


Figure 4.7 Distribution of EH-Signatures in APC queue with 14 callback functions

Moreover, there are in total 199 device drivers in our evaluation system, but our training only observes a subset of them (e.g., 51 in terms of *IopCompleteRequest*); so some legitimate KQ requests from the remaining drivers may be triggered by events such as inserting a USB device, which we have not tested yet. Fortunately, our experience suggests that it is possible to collect the set of EH-Signatures that fits the configuration and usage of a given system with enough training workloads.

4.4.5. **Performance Overhead**

We evaluate the performance overhead of KQguard in two steps: microbenchmarks and macrobenchmarks.

For the first step, we measure the overhead of KQguard validation check and heap object tracking. KQguard validation check matches Callback-Signatures against the EH-Signature Collection, and its overhead consists of matching the four parts of a Callback-Signature. Heap object tracking affects every heap allocation and deallocation operation (e.g., *ExAllocatePoolWithTag* and *ExFreePool*). These heap operations are invoked at a global level, with overhead proportional to the overall system and application use of the heap. Specifically, we measure the total time spent in performing 1,000 KQguard validation checks for the DPC queue and the I/O timer queue, two of the most active KQs.

The main result is that global heap object tracking during the experiment dominated the KQguard overhead. Specifically, DPC queue validation consumed 93.7 milliseconds of CPU, while heap object tracking consumed 8,527 milliseconds. These 1,000 DPC callback functions are dispatched over a time span of 250,878 milliseconds (4 minutes 11 seconds). Therefore, the total CPU consumed by our KQguard validation for DPC queue and the supporting heap object tracking is 8,620.7 milliseconds (or about 3.4% of the total elapsed time). The measurements of the I/O timer queue (180 ms for validation, 11,807 ms for heap object tracking, and 345,825 ms total elapsed time) confirm the DPC queue results.

For the second step, Table 4.3 shows the results of five application level benchmarks that stress one or more system resources, including CPU, memory, disk, and network. The first benchmark is Super PI, a CPU-intensive workload calculating 32 million digits of π. The second benchmark copies a directory with a total size of 1.5 GB, which stress the file system. The third and the fourth benchmarks are also CPU-intensive, performing the compression and decompression of the 1.5 GB directory with 7-Zip. The fifth benchmark downloads a 160 MB file with WinSCP, which stresses the network connection. Each workload is run multiple times and the average is reported. We can see that in terms of execution time of the selected applications, KQguard incurs modest elapsed time increases, from 2.8% for decompression to 5.6% for directory copy. These elapsed time increases are consistent with the microbenchmark measurements, with higher or lower heap activities as the most probable cause of the variations. We also run the PostMark file system benchmark and the PassMark PerformanceTest benchmark and see similar overhead (3.9% and 4.9%, respectively).

Workload	***Original (sec)***	***KQ Guarding (sec)***	***Slowdown***
Super PI [34]	2,108±41	2,213±37	5.0%
Copy directory (1.5 GB)	231±9.0	244±15.9	5.6%
Compress directory (1.5 GB)	1,113±24	1,145±16	2.9%
Decompress directory (1.5 GB)	181±4.1	186±5.1	2.8%
Download file (160 MB)	145±11	151±11	4.1%

Table 4.3 Performance Overhead of KQ Guarding in WRK

4.5. **Related Work**

In this section, we survey related work that can potentially solve the KQ injection problem and satisfy the four design requirements: efficiency, effectiveness, extensibility, and inclusiveness (Section 4.1.3).

SecVisor [65] and NICKLE [79] are designed to preserve kernel code integrity or block the execution of foreign code in the kernel. They can defeat callback-into-malware KQ attacks because such attacks require that malicious functions be injected somewhere in the kernel space. However, they cannot detect callback-into-libc attacks because such attacks do not inject malicious code or modify legitimate kernel code.

HookSafe [23] is capable of blocking the execution of malware that modifies legitimate function pointers to force a control transfer to the malicious code. However, HookSafe cannot prevent KQ injection attacks because they do not modify existing and legitimate kernel function pointers but instead supply malicious data in their own memory (i.e., the KQ request data structures).

CFI [22] can ensure that control transfers of a program during execution always conform to a predefined control flow graph. Therefore, it can be instantiated into an alternative defense against KQIs that supply malicious control data. However, CFI cannot defeat the type of KQI attacks that supply malicious non-control data because they do not change the control flow.

SBCFI [46] can potentially detect a callback-into-malware KQ attack. However, SBCFI is designed for persistent kernel control flow attacks (e.g., it only checks periodically) but KQ injection attacks are transient, so SBCFI may miss many of them. Moreover, SBCFI requires source code so it does not satisfy the inclusiveness requirement.

IndexedHooks [51] provides an alternative implementation of CFI for the FreeBSD 8.0 kernel by replacing function addresses with indexes into read-only tables, and it is capable of supporting new device drivers. However, similar to SBCFI, IndexedHooks requires source code so it does not satisfy the inclusiveness requirement.

PLCP [59] is a comprehensive defense against KQ injection attacks, capable of defeating both callback-into-malware and callback-into-libc attacks. However, PLCP does not satisfy the inclusiveness requirement due to its reliance on source code.

4.6. **Summary**

Kernel Queue (KQ) injection attacks are a significant problem. We test 125 real world malware attacks [73][76][6][78][62][63][74][77][75] and nine synthetic attacks to cover 20 KQs in the WRK. It is important for a solution to satisfy four requirements: efficiency (low overhead), effectiveness (precision and recall of attack detection), extensibility (accommodation of new device drivers) and inclusiveness (protection of device drivers with and without source code). Current kernel protection solutions have difficulties with simultaneous satisfaction of all four requirements.

We describe the KQguard approach to defend kernels against KQ injection attacks. The design of KQguard is independent of specific details of the attacks. Consequently, KQguard is able to defend against not only known attacks, but also anticipated future attacks on currently unscathed KQs. We evaluated the WRK implementation of KQguard, demonstrating the effectiveness and efficiency of KQguard by running a number of representative application benchmarks. In effectiveness, KQguard achieves very low false negatives (detecting all but two KQ injection attacks in 125 real world malware and nine synthetic attacks) and zero false positives (no false alarms after a proper training process). In performance, KQguard introduces a small overhead of about 100 microseconds per validation and up to about 5% slowdown for resource-intensive application benchmarks due to heap object tracking.

5. CONCLUSION AND FUTURE WORK

5.1. Conclusion

In this thesis, we have addressed two research problems: (1) detection of data invariants and integrity of KQ requests from a commodity operating systems kernel; (2) defense against kernel-level malware that violates these two integrity properties.

For the detection of data invariants, we develop a program analysis tool that can automatically derive data invariants from the source code of a given kernel, using static analysis. Our tool applies compiler technology to analyze the control and data flows (e.g., assignments and function calls) of a target program and reason about the global variables that are invariants. In developing this tool, we have overcome several challenges in large-scale C program analysis, such as field-sensitivity, array-sensitivity, pointer analysis, and handling of assembly code.

With this analysis tool, we are able to make a thorough study of invariants detection for the Linux kernel and the Windows Research Kernel using static analysis. To the best of our knowledge, there has not been a similar study. Both kernels are very complex software posing great challenges for static analysis by their wide use of pointers and complex structures. Our tool is able to process 400,492 lines of Linux kernel (version 2.4.32) code and identify 284,471 invariants essential to the Linux kernel's runtime integrity. To validate the result of our tool (e.g., precision), we develop a dynamic invariant detector (following the spirit of Daikon [41]) and compare it with our static analyzer. The comparison suggests that static invariant detection outperforms dynamic invariant detection in terms of false positives. For example, in the constant invariants category, we find 17,182 variables that can cause false alarms for the dynamic analyzer, while our static tool only misses 18 true invariants (with false negative rate 0.013%).

We also develop an invariant monitor based on the result of the static analysis, which detects invariant violations by ten real-world Linux rootkits and generates only one false alarm against benign workloads. Moreover, although Windows Research kernel (version Windows Server 2003), which has 665,969 lines of C code, is even more complex than the Linux kernel, our tool analyzes it successfully and detects 202,992 invariants. Comparison with the result of a dynamic analyzer shows that the dynamic analyzer generates 21,670 false constant invariants while our static tool wrongly classifies only seven true constant invariants as non-constants (with false negative rate 0.007%). We develop an invariant monitor in the same way as the Linux kernel and this monitor successfully detects nine real-world Windows malware samples and one synthetic Windows malware, while emitting only one false alarm. Our study shows that our static tool is an effective tool for a system security expert. Furthermore, our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can potentially have very low false positive and false negative rates.

On the other hand, we first build a static analysis tool to automatic detect KQs from kernel source code. As a result, we found found 20 KQs in the Windows Research Kernel and 22 in Linux kernel. We then figure out the KQguard mechanism that can decide the legitimacy of a KQ request (e.g., we can distinguish the KQ requests made by a rootkit from the legitimate ones). Due to the lack of source code of many third-party device drivers, we apply dynamic analysis of the binary code for the automatic generation of the specifications of legitimate KQ requests (called EH-Signatures). In this way, we avoid the requirement of source code of the device drivers and satisfy the inclusiveness.

Based on the KQguard mechanism and the EH-Signatures generated, we implement KQguard in WRK and the Linux kernel. We further evaluate its effectiveness against KQ exploits. First, we chose 125 malware samples from the top malware families and top botnet families and create 9 synthetic rootkits. KQguard is able to detect all except two KQ injection attacks, which means it has very low false negatives. Second, after a proper training, KQguard can have no false alarms for representative workloads. The very low false negatives and no false alarms together demonstrate the effectiveness of KQguard. The performance evaluation shows that for resource intensive benchmarks, KQguard introduces a small overhead of up to about 5%, which illustrates the efficiency of KQguard. In KQguard design we isolate the legitimate KQ request into a table (EH-Signature collection) that can be easily extended in a training phrase, which achieves the goal of extensibility.

Unlike the various signature-based detection approaches, our defense not only works for existing malware but also can handle future malware. Specifically, our defense can detect a kind of malware that temper with invariants or adopt KQI attacks. To catch up with numerous new malware, most existing commercial anti-virus software need to update their malware signature database very frequently. Even though, they still can only detect existing malware whose signature is known. While our defense does not need updates unless the system kernel is changed or there is a new legitimate device driver, and can also deal with future malware.

5.2. **Future Work**

In this thesis, we protect the detected invariants with our Invariant Monitor that checks the invariants periodically (every 30 seconds). However, such kind of protection can possibly be defeated by a transient attack---an attack who knows the check mechanism can modify the value of an invariant after one check and restore its original value before next check. One mitigated solution to such attack is to randomize the interval between two checks (e.g., the check interval can vary from 1 second to 60 seconds).

Another solution is to leverage the hardware-based protection so that any write-access to the invariants can be monitored. However, nowadays the hardware can only provide page-level protection, which means that if we want to write-protect an invariant, the memory page that contains the invariant will be also write-protected. While in a commodity OS kernel such as Linux and Windows, it is very common that there exist hundreds of thousands of invariants (as our detection results in Section 3.2) and these invariants are widely scattered cross the kernel space. And it is normal that the invariants can co-locate with some frequently modified variables together in the same page. Suppose we deploy the approach that we write-protect all pages that contain the invariants and verify all write-accesses to these pages (whether the write-access is applied to an invariant), it will cause significant performance overhead, which is mostly introduced from the unnecessary page faults that are triggered by write-accesses to non-invariants.

We recognize this protection granularity gap that hardware can only provide page-level protection but invariant protection needs byte-level granularity. To address this challenge, we can adopt a solution similar to HookSafe [23] that we relocate invariants to a dedicated page-aligned memory region and then introduce an indirection layer to manage accesses to them with hardware-based page-level protection. Specifically, the access to the original invariant will be redirected to the relocated one by the indirection layer. In this way, we can avoid the unnecessary page faults triggered by the write-accesses to irrelevant non-invariants.

Our current invariant analysis is only applied on the global variables---more specifically, statically allocated variables. However, except these global variables, there also exist many kernel objects that are dynamically allocated from heap at runtime. During our study of kernel queues, we found that the life time of some heap objects can last from the time they are allocated till when the system is shut down. So we can view this kind of heap objects as global variables. Furthermore, this kind of heap objects can also be critical to system security. For example, we found there are eight elements in the I/O timer queue at runtime and the callback function pointers in the eight elements do not change once the elements are inserted, which implies that we can apply invariant analysis to derive such heap invariants.

Since our static detection of data invariants needs source code, it will not work for the closed-source kernel. And it cannot handle assembly code without human aid. We can improve our approach by applying the static binary analysis techniques such as Vine in BitBlaze [39] and BAP [40], which provides an infrastructure for analyzing binary.

Our invariant detection has false positives due to the imprecision of our pointer analyzer based on the generalized one level flow (GOLF) algorithm. We can build a more precise pointer analyzer based on the Anderson's algorithm [26].

Nowadays, the smartphone become more and more popular in our daily life and it also has an operating system with a Linux kernel, which means that it is possible for us to apply our system on it and derive data invariants and integrity of KQs and further defend against the malware that tries to exploit them.

REFERENCES

[1] eBay. http://www.ebay.com/

[2] Facebook. https://www.facebook.com/

[3] Butler, J. DKOM (Direct Kernel Object Manipulation). http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf.

[4] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants", In ACSAC '08, pages 77–86. IEEE Computer Society, 2008.

[5] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence", DSN'09, Lisbon, Portugal, 2009.

[6] Decker, A., Sancho, D., Kharouni, L., Goncharov, M., and McArdle, R. Pushdo/Cutwail: A Study of the Pushdo/Cutwail Botnet. Trend Micro Technical Report, 05/2009

[7] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, C. D. McDonell, "Linux kernel integrity measurement using contextual inspection", 2007 ACM workshop on STC. October 2007.

[8] P. Barham, B. Dragovic, K. Fraser, et al., "Xen and the art of virtualization", ACM SOSP, Bolton Landing, NY, Oct. 2003

[9] F. Bellard, "QEMU, a fast and portable dynamic translator", Proceedings of the 2005 USENIX Annual Technical Conference, 2005.

[10] N. Petroni, Jr., T. Fraser, J. Molina, W. A. Arbaugh, "Copilot—a coprocessor-based kernel runtime integrity monitor", 13th USENIX Security Symposium, San Diego, CA, Aug. 2004.

[11] Trusted Platform Modules. http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications.

[12] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence", DSN'09, Lisbon, Portugal, 2009.

[13] T. Garfinkel, M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection", NDSS, February 2003.

[14]Chkrootkit - rootkit detection tool v0.49. http://www.chkrootkit.org.

[15]RootkitRevealer. http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx

[16]Sophos anti-rootkit. http://www.sophos.com/products/freetools/sophos-anti-rootkit.html

[17]Yi-Min Wang, Roussi Roussev, Chad Verbowski, Aaron Johnson, Ming-Wei Wu, Yennun Huang, and Sy-Yen Kuo. "Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management". In LISA '04.

[18]A. Baliga, P. Kamat and L. Iftode, "Lurking in the shadows: identifying systemic threats to kernel data", IEEE S&P, Oakland, CA, May 2007

[19]Microsoft. Windows Research Kernel v1.2. https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7366&c1=en-us&c2=0

[20]Brumley, D. 1999. Invisible intruders: rootkits in practice. ;login:, 24, Sept. 1999

[21]Hoglund, G. 2006. Kernel Object Hooking Rootkits (KOH Rootkits). http://my.opera.com/330205811004483jash520/blog/show.dml/314125.

[22]Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. 2005. Control flow integrity. Proceedings of the 12th ACM Conference on Computer and Communications Security.

[23]Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection", Proceedings of ACM Conference on Computer and Communications Security (CCS '09), 2009.

[24]N. Petroni, T. Fraser, A. Walters, and W. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data", 15th USENIX Security Symposium, 2006.

[25]http://users.cis.fiu.edu/~weijp/Jinpeng_Homepage_files/report.xml (It is a huge file. Please open it with a text editor instead of the browser).

[26]L. O. Anderson, "Program analysis and specialization for the C programming language", PhD thesis, University of Copenhagen, 1994.

[27]David Wagner. "Static analysis and computer security: New techniques for software assurance", Ph.D. dissertation, Dec. 2000, UC Berkeley.

[28]Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof, "Estimating the Impact of Scalable Pointer Analysis on Optimization", In SAS'01, London, UK, 2001.

[29]Microsoft MSDN. http://msdn.microsoft.com.

[30]G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. “CIL: Intermediate language and tools for analysis and transformation of C programs”, Conference on CC, Grenoble, France, Apr. 2002.

[31]Linux Test Project, version 20050107. http://ltp.sourceforge.net.

[32]Iperf v2.0.5. http://sourceforge.net/projects/iperf.

[33]Andrew benchmark, reinterpreted version by Yasushi Saito, HP Labs, Storage Systems Department. http://www.ysaito.com/andrew-tcl-bench-0.3.tgz.

[34]Super PI v1.5. http://www.superpi.net.

[35]Iozone v3.405. http://www.iozone.org.

[36]7zip v9.20. http://www.7-zip.org.

[37]Windows Driver Kit v7.1.0. http://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx.

[38]Tom Espiner. “’Storm Worm’ Slithers on”, http://www.zdnet.com/storm-worm-slithers-on-3039285565.

[39]Dawn Song, et al. “BitBlaze: a new approach to computer security via binary analysis”, Information systems security. Springer Berlin Heidelberg, 2008, pp 1-25.

[40]David Brumley, et al. “BAP: a binary analysis platform”, Computer Aided Verification. Springer Berlin Heidelberg, 2011.

[41]M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants”, In Science of Computer Programming, 2007.

[42]Radu Rugina and Martin Rinard, “Symbolic bounds analysis of pointers, array indices, and accessed memory regions”, PLDI’00, pages 182-195, 2000.

[43]David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, “A First Step Toward Automated Detection of Buffer Overrun Vulnerabilities”, Proceedings of the Network and Distributed System Security Symposium, San Diego, California, February 2000.

[44]R. Sailer, X. Zhang, T. Jaeger, L. van Doorn, “Design and implementation of a TCG-based integrity measurement architecture”, 13th USENIX Security Symposium, 2004.

[45]L. Davi, A. Sadeghi, and M. Winandy, “Dynamic Integrity Measurement and Attestation: Towards Defense against Return-Oriented Programming Attacks”, 2009 ACM workshop on STC. November 2009.

[46]N. Petroni and M. Hicks, “Automated detection of persistent kernel control-flow attacks”, 14th ACM CCS, Alexandria, VA, Oct. 2007.

[47]X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, “Secure coprocessor-based intrusion detection”, Tenth ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002.

[48]Martim Carbone, et al. “Mapping kernel objects to enable systematic integrity checking”, Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.

[49]M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity”, ACM Conference on CCS, Alexandria, VA, Nov. 2005.

[50]T. Jaeger, R. Sailer, and U. Shankar, “PRIMA: policy-reduced integrity measurement architecture”, SACMAT 2006.

[51]J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, “Comprehensive and Efficient Protection of Kernel Control Data”, IEEE Transactions on Information Fo-rensics and Security, 6(2), June 2011.

[52]J. Grizzard, E. Dodson, G. Conti, J. Levine, and H. Owen, “Toward a trusted immutable kernel extension (TIKE) for self-healing systems: a virtual machine approach”, 5th IEEE IAW, 2004.

[53]J. Levine, J. Grizzard, and H. Owen, “Re-establishing trust in compromised systems: recovering from rootkits that trojan the system call table”, Proceedings of 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 2004.

[54]Mihai Christodorescu, et al. “Semantics-aware malware detection”, Security and Privacy, 2005 IEEE Symposium on. IEEE, 2005.

[55]Trusted Computing Group. http://www.trustedcomputinggroup.org.

[56]W. A. Arbaugh, D. J. Farber, and J. M. Smith, “A secure and reliable bootstrap architecture”, In IEEE S&P 1997, pp. 65–71.

[57]J. Sheehy, G. Coker, J. Guttman, et al. “Attestation: evidence and trust”, Mitre Technical Paper, 2007. http://www.mitre.org/publications/technical-papers/attestation-evidence-and-trust, accessed Jan 5, 2014.

[58]L. Singaravelu, C. Pu, H. Haertig, C. Helmuth, “Reducing TCB complexity for security-sensitive applications: three case studies”, Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, Leuven, Belgium, April 2006.

[59]Wei, J., and Pu, C. 2012. Towards a General Defense against Kernel Queue Hooking Attacks. Elsevier Journal of Computers & Security, Volume 31, Issue 2, pp. 176-191.

[60]Microsoft. Using Timer Objects. http://msdn.microsoft.com/en-us/library/ff565561.aspx

[61]Chiang, K., Lloyd, L. 2007. A Case Study of the Rustock Rootkit and Spam Bot. Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots'07).

[62]Kapoor, A. and Mathur, R. 2011. Predicting the future of stealth attacks. Virus Bulletin 2011, Barcelona.

[63]Kaspersky Lab. The Mystery of Duqu: Part Five. http://www.securelist.com/en/blog/606/The_Mystery_of_Duqu_Part_Five.

[64]Symantec Connect Community. W32.Duqu: The Precursor to the Next Stuxnet. Oct. 2011. http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet, accessed April 2013.

[65]Seshadri, A., Luk, M., Qu, N., and Perrig, A. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In Proceedings of ACM SOSP’07.

[66]Solar Designer. Bugtraq: Getting around non-executable stack (and fix). Website. http://seclists.org/bugtraq/1997/Aug/63.

[67]Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V. W., and Ning, P. 2011. On the Expressiveness of Return-into-libc Attacks. In Proceedings of RAID 2011.

[68]PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, accessed April 2013

[69]Necula, G. C., McPeak, S., Rahul, S. P. and Weimer, W. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. Proceedings of Conference on Compiler Construction (CC), Apr. 2002.

[70]Wei, J., Zhu, F., and Pu, C. 2012. KQguard: Protecting Kernel Callback Queues. Florida International University Technical Report, TR-2012-SEC-03-01. http://www.cis.fiu.edu/~weijp/Jinpeng_Homepage_files/WRK_Tech_Report_03_12.pdf

[71]Top 20 Malware Families in 2010. http://blog.fireeye.com/research/2010/07/worlds_top_modern_malware.html.

[72]Top 10 Botnet Families in 2009. https://blog.damballa.com/archives/572.

[73]Anselmi, D., et al. 2011. Battling the Rustock Threat. Microsoft Security Intelligence Report, Special Edition, January 2010 through May 2011.

[74]Kwiatek, L. and Litawa, S. Yet another Rustock analysis... Virus Bulletin, August 2008.

[75]Prakash, C. 2008. What makes the Rustocks tick! Proceedings of the 11th Association of anti-Virus Asia Researchers International Conference (AVAR'08).

[76]Boldewin, F. 2007. Peacomm.C - Cracking the nutshell. Anti Rootkit, September 2007.
http://www.antirootkit.com/articles/eye-of-the-storm-worm/Peacomm-C-Cracking-the-nutshell.html.

[77]OffensiveComputing. Storm Worm Process Injection from the Windows Kernel. http://offensivecomputing.net/papers/storm-3-9-2008.pdf

[78]Giuliani, M. ZeroAccess – an advanced kernel mode rootkit, rev 1.2. www.prevxresearch.com/zeroaccess_analysis.pdf

[79]Riley, R., Jiang, X., and Xu, D. 2008. Guest-transparent prevention of kernel rootkits with VMM-Based memory shadowing. Proceedings of RAID'08.

VITA

FENG ZHU

2005-2009	B.A., Computer Science Nanjing University Nanjing, China
2009-2014	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Jinpeng Wei, Calton Pu, Carlos V. Rozas, Anand Rajan, and Feng Zhu (2010). *Modeling the Runtime Integrity of Cloud Servers: a Scoped Invariant Perspective.* International Workshop on Cloud Privacy, Security, Risk and Trust (CPSRT 2010). In conjunction with the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010), Indianapolis, IN, Nov. 30 - Dec. 3, 2010.

Jinpeng Wei, Feng Zhu, and Yasushi Shinjo (2011). *Static Analysis Based Invariant Detection for Commodity Operating Systems.* 7th International Conference on Collaborative Computing (CollaborateCom 2011), Orlando, FL, October 15-18, 2011.

Jinpeng Wei, Calton Pu, Carlos V. Rozas, Anand Rajan, and Feng Zhu (2013). *Modeling the Runtime Integrity of Cloud Servers: a Scoped Invariant Perspective.* Privacy and Security for Cloud Computing. Springer London, 2013. 211-232.

Jinpeng Wei, Feng Zhu, and Calton Pu (2013). *KQguard: Binary-Centric Defense against Kernel Queue Injection Attacks.* Computer Security–ESORICS 2013. Springer Berlin Heidelberg, 2013. 755-774.

Feng Zhu and Jinpeng Wei. (2014). *Static Analysis Based Invariant Detection for Commodity Operating Systems*. Computer & Security, 43 (2014):49-63.