Towards a General Defense against Kernel Queue Hooking Attacks

Jinpeng Wei[1] and Calton Pu[2]

[1]Florida International University, 11200 SW 8th Street, Miami, FL, 33199 USA, weijp@cs.fiu.edu

[2]Georgia Institute of Technology, 266 Ferst Dr, Atlanta, GA 30332 USA, calton@cc.gatech.edu

Abstract

Kernel queue hooking (KQH) attacks achieve stealthy malicious function execution by embedding malicious hooks in dynamic kernel schedulable queues (K-Queues). Because they keep kernel code and persistent hooks intact, they can evade detection of state-of-the-art kernel integrity monitors. Moreover, they have been used by advanced malware such as the Rustock spam bot to achieve malicious goals. In this paper, we present a systematic defense against such novel attacks. We propose the Precise Lookahead Checking of function Pointers approach that checks the legitimacy of pending K-Queue callback requests by proactively checking function pointers that may be invoked by the callback function. To facilitate the derivation of specifications for any K-Queue, we build a unified static analysis framework and a toolset that can derive from kernel source code properties of legitimate K-Queue requests and turn them into source code for the runtime checker. We implement proof-of-concept runtime checkers for four K-Queues in Linux and perform a comprehensive experimental evaluation of these checkers, which shows that our defense is effective against KQH attacks.

Keywords: control flow integrity; kernel queue hooking; rootkits; runtime defense; static analysis

1 Introduction

Rootkits have become one of the most dangerous threats to systems security. Because they run at the same privilege level as the operating systems kernel, they can modify the OS behavior

in arbitrary ways. For example, they can tamper with existing code and/or data of the OS to conceal the runtime state of the system in terms of running processes, network connections, and files. Besides, they can add new functionalities to the OS to carry out malicious activities such as key logging and sensitive information collection. In recent years, significant work has been proposed to detect [1,2,3,4,5,6,7,8,9] analyze [10,11,12,13,14], or defend against [15,16,17,18] rootkits.

However, existing work on rootkits mainly focuses on attacks that *change* legitimate kernel code [6,15,16,17] or *change* legitimate kernel hooks (locations in kernel space that hold function pointers) [7,18], but falls short of attacks that *create* malicious hooks in dynamically allocated kernel objects, as demonstrated by kernel queue hooking (KQH) attacks (Section 2). Briefly speaking, KQH rootkits leverage various callback mechanisms of the kernel, which enable the rootkits to direct kernel control flow as effectively as exploiting buffer overflows, and by reusing legitimate kernel code, these rootkits can successfully hijack control flow of the victim kernel, yet remain invisible to state-of-the-art defense techniques. Specifically, KQH attacks are unique in three important ways:

- They do not hijack existing, legitimate kernel hooks; instead, they create their own malicious kernel hooks.
- They leverage kernel data structures that can have multiple instances at the same time. For example, although the Linux kernel allows each type of interrupt to have only one handler registered in the interrupt descriptor table (IDT), it allows *multiple IRQ action handlers* for the same interrupt to coexist.
- They leverage dynamic kernel data structures. Again take IRQ action handlers as example, the kernel uses a queue to keep track of currently registered IRQ action handlers

and this queue can grow or shrink at runtime depending on which entities are interested in a particular IRQ, including rootkits.

Therefore, it is much harder to detect malicious manipulations of the IRQ action queue than those of the IDT because it is non-trivial to find the *known-good* values for the IRQ action queue; and it is also much harder to defend against KQH attacks on the IRQ action queue. For example, we cannot simply make the embedded hooks immutable (as proposed by HookSafe[18]), because we do not know whether a hook is benign or malicious to start with.

For ease of presentation, we call data structures such as the IRQ action queue *kernel schedulable queues* (or K-Queues for short), and attacks that insert malicious requests to such queue-like data structures *K-Queue hooking* (KQH) attacks. We elaborate on the difference between KQH attacks and other attacks in Section 3.2.

Moreover, as we will discuss in Section 2.2, advanced and real malware is already misusing K-Queues to their advantage; examples of such malware include major spam bots such as Rustock, Pushdo/Cutwail, and Storm/Peacomm.

Therefore, the technical novelty and the realistic threat of KQH attacks call for new defense approaches. The challenge is that malicious data objects share the same K-Queues with legitimate data objects, and both kinds can be created and destroyed at runtime. So a reasonable defense has to check the legitimacy of each object each time the kernel uses it for control transfer decisions (because such objects may not be persistent). To this end, we design and implement a hypervisor-supported reference monitor that *intercepts* and *rejects* malicious callback requests while allowing legitimate kernel callback requests to proceed. Secondly, it is very subtle and tricky to develop a specification for legitimate kernel data objects. Manually doing this for a code base as large as the Linux kernel is hopeless and error-prone at best. To address this issue,

we build automated tools to derive such specifications more efficiently. Moreover, we employ code generation techniques to automatically translate the inferred specifications into runtime check code.

Specifically, we employ static program analysis (e.g., points-to analysis and transitive closure analysis) of the kernel source code to derive legitimacy specifications. To facilitate the generation of specifications for the entire class of K-Queues, we build a unified static analysis framework and a set of tools that can not only derive specifications from kernel source code, but also generate the corresponding checking code for different K-Queue instances. Furthermore, we design and implement a proof-of-concept runtime reference monitor that checks pending requests for four types of K-Queues (IRQ action queue, tasklet queue, soft timer queue, and task queue) in the Linux kernel and guards the check result against tampering. Finally, we perform a comprehensive experimental evaluation of our tools as well as a detailed performance overhead analysis of the reference monitor, which shows that our approach is able to detect all sample KQH rootkits that we have.

The rest of this paper is organized as follows. Section 2 introduces KQH attacks and the threats they pose to systems security. Section 3 presents our modeling of KQH attacks and a high level description of our defense. Section 4 discusses our design of a unified static analysis framework and tools that can generate specifications of legitimate K-Queue requests and turn them into checking code. Section 5 presents our implementation of the defense. Section 6 discusses evaluation of our defense, Section 7 talks about related work, and Section 8 concludes the paper.

2 Kernel Queue Hooking

Kernel Queue Hooking (KQH) is a rootkit technique that leverages extensible kernel data structures, such as queues, to inject malicious control flows into the victim kernel. The *queues* contain instances of dynamic kernel data structures and they can be *hooked* because they are organized in linked lists. Finally, running malicious logic is possible because such data structures contain function pointers.

2.1 Kernel Queue Hooking Attacks

For example, Figure 1 shows the tasklet queue in Linux kernel 2.4.32, which is used by the kernel to support deferred code execution (e.g., the bottom-half of an interrupt handler routine). The type definition for each component in this queue is shown in Figure 3. We can see that each component structure of the tasklet queue contains a function pointer (in the `func` field), contextual information (in the `data` field), and a pointer (in the `next` field) to the next component in the queue. This data structure represents a request for the kernel to invoke the callback function whose address is stored in the `func` field (e.g., `kbd_bh` in Figure 1) and pass along the `data` field as an input parameter. Tasklets can be requested by any kernel component through the `tasklet_schedule` or `tasklet_hi_schedule` APIs.

To mount a KQH attack via the tasklet queue, a rootkit can initialize a tasklet structure including the `func` field and the `data` field, and call `tasklet_schedule` or `tasklet_hi_schedule` to add this request into the tasklet queue (the rootkit can also add its request by directly manipulating the tasklet queue, but this makes no difference to our defense). When the kernel handles the rootkit's request, the callback function in the `func` field is invoked with the `data` field as an input parameter. Since the `func` and `data` fields are supplied by the rootkit, the rootkit can direct kernel control flow to its desired location.

To some extent, a K-Queue provides the same opportunity for malware as a buffer overflow: to gain illicit control over the kernel execution flow. The difference is that K-Queue hooking uses the callback function field of a K-Queue request, while a buffer overflow attack uses the return address on the stack. A rootkit can directly prepare a malicious function and set its address in a K-Queue request, or set the address of a legal kernel function in a malicious K-Queue request, in a way similar to a "return-to-libc" [19] or return oriented programming (ROP) [20,21] attack.

To illustrate the feasibility of the "return-to-libc" or ROP style K-Queue hooking, we develop a prototype keylogger that depends only on legitimate kernel code for its normal operation. Specifically, the kernel function `dump_regs` (in linux-kernel-2.6.16/drivers/block/umem.c) takes one input parameter (a pointer to a structure of type `cardinfo`) and dumps the first 128 bytes of a character array pointed to by the `csr_remap` field of the `cardinfo` structure. Our prototype keylogger requests a soft timer [22] (an instance of K-Queue) with `dump_regs` as the callback function and a fabricated pointer as the data field, whose type is `struct cardinfo*` and whose `csr_remap` field contains the address of a keyboard input buffer (that of /dev/tty7). When the soft timer expires, the kernel invokes `dump_regs`, and because of the setting of the contextual data by the rootkit, `dump_regs` dumps the first 128 bytes of the keyboard input buffer into the system log file. To achieve periodic keyboard input buffer dumping, multiple soft timers are requested at once. These soft timer requests are the same except for expiration time, which causes the kernel to invoke `dump_regs` periodically. We have implemented such a keylogger as a loadable kernel module for Linux kernel 2.6.16, and the requests for all the soft timers are done in the initialization function of the kernel module.

More generally, a rootkit can "schedule" a sequence of legitimate kernel calls to serve its malicious purpose using a sequence of K-Queue requests. Suppose the rootkit wants to invoke a sequence of calls to legitimate kernel functions $f_1, f_2, \cdots, f_n$, and each function takes one parameter as input, the rootkit can request *n* soft timers in which timer *i* has f_i as its callback function and a_i as its contextual data, and the timers expire in their numerical order. Then the kernel would execute the $f_1(a_1), f_2(a_2), \cdots, f_n(a_n)$ sequence for the rootkit as the *n* timers expire one after the other. The point here is that *a rootkit does not have to inject malicious code into the kernel to satisfy its needs*. Fortunately, our defense is able to address this type of rootkits.

2.2 KQH Rootkits in the Wild

Advanced and real malware is actively misusing K-Queues to their advantage. One use is for self-protection. For example, the Rustock.C spam bot relies on two Windows kernel Timers [23] to check whether it is being debugged/traced [24] (e.g., `KdDebuggerEnabled` is true). Another use is to monitor the events on the victim system. For example, Pushdo/Cutwail botnet sets up three callback routines by invoking `IoRegisterFsRegistrationChange`, `CmRegisterCallback`, and `PsSetCreateProcessNotifyRoutine`, respectively [25]; these callback routines enable it to monitor file system registrations, monitor, block, or modify a registry operation, and inject a malicious module into a `services.exe` process. A third misuse of K-Queues is to disable security products, which is exemplified by the Storm/Peacomm spam bot that invokes `PsSetLoadImageNotifyRoutine` to register a malicious callback function that disables security products [26]. Some malware even relies on kernel queues for its normal functioning. For example, the Rustock.C spam bot uses APCs (Asynchronous Procedure Calls) to send spam messages [27].

The above real-world malware all hook K-Queues in the Windows kernel, which means that KQH attacks on Windows kernels are realistic. Then one interesting question is whether similar KQH malware exists for other kernels such as Linux and Mac OS. We have experimentally confirmed the effectiveness of hooking the soft timer queue of the Linux kernel by rootkits [22]. More theoretically, KQH attacks are quite feasible because of the large number of kernel data structures that can be considered K-Queues. Figure 5 lists some example K-Queues in Linux and Windows kernels. Structural details of three kinds of queues in Linux are shown in Figure 2 (IRQ action queue), Figure 3 (tasklet queue), and Figure 4 (task queue).

3 KQH Threat Modeling and Defense Ideas

3.1 Threat Modeling

We adopt the standard queuing model for K-Queues. That is, each K-Queue contains a request queue and a server. A request is inserted into the queue by an enqueue operation and it is removed from the queue by a dequeue operation by the server. This model works well for the K-Queues. For example, all instances of K-Queues provide APIs (the enqueue operations) for submitting an execution request for some callback function, and they all have a dispatch engine (the server) that takes the request from the queue (the dequeue operation) and invokes the callback function.

KQH attacks are a trust issue because existing K-Queue servers simply trust whatever request found in the request queue. I.e., once a request enters a queue, it is fully trusted by the corresponding K-Queue server. This is fine if we assume that the requests all come from trusted kernel components. However, this assumption no longer holds when we take malware into consideration. For example, a rootkit can insert malicious requests that contain function pointers and contextual data of the attacker's choosing.

3.2 Defense Idea

3.2.1 Motivation of the Defense

Compared with traditional kernel hook hijacking [18], in which a rootkit replaces kernel hooks (existing and legitimate function pointers) with malicious function pointers, KQH is a stealthier attack scheme. Kernel hook hijacking is invasive because the malware does not own the *already-existing* kernel objects (e.g., the IRP function table of a Windows device driver like tcpip.sys and disk.sys) that it manipulates. This implies two things: (1) the manipulations can be safely treated as interference or anomaly, and (2) it is feasible to lock up the innocent kernel hooks to prevent hijacking, as implemented by HookSafe [18]. On the other hand, KQH is non-invasive because the K-Queue data structure that holds the malicious request is created thus owned by the malware and the attack does not rely on overwriting any existing kernel hook or function pointer. Therefore, KQH attacks are beyond the reach of HookSafe. Second, Kernel hook hijacking makes *persistent* changes to kernel control flow, but KQH is capable of injecting *transient* control flows. Moreover, as demonstrated by the keylogger that we discuss in Section 2, a KQH rootkit does not have to inject malicious code for its normal execution. Therefore, this kind of rootkits cannot be completely prevented by tools such as SecVisor[17] and NICKLE[16], which block the execution of injected code in the kernel. CFI [28] can check the integrity of all function pointers within a program, so theoretically it can be instantiated into a checker against KQH attacks. However, an implementation of the original CFI design for the Linux kernel is not available to the best of our knowledge. SBCFI [7] is a control flow integrity checker for the Linux kernel that takes the spirit of CFI but has a different way of checking: instead of checking each function pointer right before it is invoked, SBCFI periodically checks *all reachable function pointers (from global variables)* in a batch. SBCFI relies on accurate type information to reach potential function pointers, but K-Queue data structures use opaque fields (e.g., void * data in

Figure 4); so SBCFI may not be able to cover every function pointer that occur in a K-Queue callback invocation (e.g., in a type 2 attack in Section 3.2.3). Moreover, the periodic nature of SBCFI checking is vulnerable to a Time-Of-Check-To-Time-Of-Use race condition [29]. However, we respectfully admit that CFI and SBCFI are powerful and general defense techniques against control flow integrity attacks. Finally, one necessary step for KQH rootkits is to insert the malicious requests into K-Queues, which seems to require running some malicious code in the kernel; if that is the case, techniques such as SecVisor[17] and NICKLE[16] would be able to stop the malware. Unfortunately, an attacker can use `/dev/kmem` (or `/dev/mem`) or a DMA-capable device [30] to directly modify kernel memory, including K-Queues, without running any code in the kernel. Because of the limitations or unavailability of existing defense techniques, we build our own detection and prevention technique in this paper.

3.2.2 Overview of the Defense

To counter KQH attacks that leverage K-Queues, we propose to intercept and check the legitimacy of K-Queue requests *before* service, so that malicious requests can be denied. There are two ways that we can do this: intercept when a request *enters* a K-Queue, or intercept when a request *leaves* a K-Queue. It would be very hard to implement the former because malware can directly manipulate the K-Queue data structures to add new requests (i.e., without using the provided enqueue APIs). On the other hand, all requests in a K-Queue must go through the same "exit" because that's where they can be served by the K-Queue dispatcher. Therefore, checking a request when it leaves a K-Queue is enough to intercept all malicious requests. Specifically, we add a *guard* into a K-Queue server which checks the legitimacy of each pending request before it leaves the K-Queue (to be served), and the guard makes its decision according to *security specifications*.

The security specifications are tasked with differentiating legitimate K-Queue requests from malicious ones, such that only the former are acted upon by the server. In this paper, the specifications are about runtime control flow integrity of the kernel. Our security specification says that *a legitimate K-Queue callback function and all functions that it calls transitively should always conform to a predetermined control flow graph*. Since direct modifications of the static kernel code segment are straightforward to detect (e.g., by hashing), the major threat to the control flow integrity of a K-Queue callback function is manipulations of function pointers involved in the K-Queue callback – such function pointers widely exist in the kernel and can be manipulated to point to anywhere that the malware wants. Therefore, our algorithm must check the legitimacy of function pointers involved in a K-Queue callback.

Different heuristics have been used in previous work to check the runtime legitimacy of function pointers. For example, a range checker considers a function pointer "good" if its target address falls within some range. However, such heuristics are not accurate because they allow malware to crash the kernel or cause subtle anomalies in the kernel by setting a function pointer to a legitimate but unexpected kernel function. Therefore, a more accurate specification is needed. Ideally, such a specification would consider a function pointer "good" only if there exists a code path in the uncompromised kernel where the target address is assigned to the function pointer. We can approximate this requirement by performing points-to analysis [31] of function pointers. In this paper, we apply static analysis technique on the kernel source code.

Specifically, our algorithm is called precise, lookahead checking of function pointers (PLCP). PLCP is *precise* because it first collects the complete set of functions that can be assigned to a function pointer and then uses that knowledge to check the integrity of a function pointer at runtime. PLCP needs to perform *lookahead* checks because it is not enough to only check the

immediate function pointer (i.e., the callback function field of a K-Queue request data structure) - that callback function may invoke other function pointer(s) “down the road”. Therefore, such function pointers need to be proactively identified and checked in advance.

3.2.3 Defense Algorithm: PLCP

Based on the way that kernel treats K-Queue requests, there can be two types of malicious K-Queue requests: type 1 supplies a malicious callback function, while type 2 supplies a legitimate callback function but a malicious data pointer. In order to detect a type 1 malicious request, we only need to check its callback function against a white list of legitimate callback functions. However, this check is inadequate for a type 2 malicious request because its immediate callback function is a legitimate one, but its data pointer can induce the control flow of the callback function maliciously, similar to a “return-to-libc” style attack [19]. Figure 6 illustrates a type 2 malicious soft timer request (in shaded color). This figure shows that the `tx_timeout` field of the data structure (in shaded color) referenced by the `data` pointer of the malicious soft timer request is set to a malicious function (e.g., `malicious_foo`), and because `dev_watchdog` (a legitimate callback function) invokes such a function pointer, it transfers control to the malware eventually. Therefore, in addition to checking the callback function, we also need to compare the `tx_timeout` field against a white list of legitimate functions (for example `e1000_tx_timeout`) that can be assigned to this field in the legitimate requests.

This case study highlights the critical parts of our PLCP algorithm as shown in Figure 7. It first checks whether the top-level callback function *cb* is legitimate by comparing it against the functions in the points-to set of structure *sn* and field *fn*. If there is a match, it further checks the legitimacy of all function pointers that can be invoked by *cb* (i.e., in safeToExec(*cb*, *d*)). This kind of checking continues recursively into the legitimate functions identified along the way, and

it terminates when the last legitimate function does not invoke any new function pointers. The details of PLCP, including points-to sets and necessary recursive checking steps, are derived by an offline static analysis of the kernel source code, including points-to analysis and transitive closure analysis, which will be discussed in Section 4.

3.3 Defense Architecture and Assumptions

Our defense directly inserts KQH guards (Section 3.2.2) in the kernel. Specifically, we statically modify the K-Queue servers of the kernel (at the source code level) so that before invoking a callback function, a K-Queue server diverts execution to the corresponding KQH guard; and the K-Queue server invokes the callback function only if the response from the K-Queue guard is positive. The information given by the K-Queue servers to the KQH guards includes the pending callback function (*cb*) and the contextual data (*d*), and the KQH guards implement the PLCP algorithm.

Because KQH attacks are at the kernel-level, a defense mechanism purely inside the same kernel would be vulnerable to tampering by an attacker; our K-Queue guards are no exception. Consequently, we put the kernel under protection in a guest virtual machine (VM) on top of the Xen [32] virtual machine manager (VMM), and we extend the shadowing-based memory management of Xen (Section 5.3) to ensure that the memory pages where our KQH guards reside are read-only, so that they cannot be maliciously modified by the attacker.

An attacker may also mount a TOCTTOU (Time-Of-Check-To-Time-Of-Use) attack [29] on our defense. This is because our defense assumes that the checked conditions hold throughout the execution of the callback function. However, this assumption can be violated if the guest kernel is multi-threaded; specifically, right after the K-Queue request is checked, but before the K-Queue callback function finishes, a malicious control flow in the guest kernel can potentially

modify a function pointer already checked, so that the callback function transfers control to where the malware chooses. This constitutes a TOCTTOU race condition. TOCTTOU attacks are non-deterministic, since the attack result depends on the interleaving of the attacker's actions (e.g., modifying a function pointer) and the victim's actions (e.g., checking the value of the function pointer and loading that value into the program counter): in order to succeed, the attacker's action must happen between the checking and loading steps of the victim, which is called the vulnerability window. For our K-Queue defense, the vulnerability window only spans the execution of a callback function, so it may be too narrow for an attack to succeed. However, the success probability can be increased significantly on multiprocessors, based on our past research [33]. Therefore, we address TOCTTOU attacks in our K-Queue defense.

To counter TOCTTOU attacks against our defense, we protect the checked function pointers from tampering during the execution of the K-Queue callback function. Specifically, during the K-Queue request checking, each participating structure field is first write-protected and then checked. When the check fails at any point, the already protected structure fields are unlocked. If the check succeeds, the unlocking is deferred until the callback function has finished. We implement the memory protection support in the underlying VMM (Section 5.3).

Our defensive mechanism makes the following assumptions. First, the hardware and the VMM are part of the TCB (trusted computing base). Second, the legitimate kernel code in the guest VM's memory, including the inserted KQH guards, is tamper-proof. This implies that the symbol-to-runtime-address mapping can be trusted in the guest kernel. In other words, a rootkit cannot overwrite a legitimate kernel function already in memory with its own malicious version. However, the rootkit may spoof legitimate functions that have not been loaded in the memory (e.g, those in loadable kernel modules or LKMs). To prevent this kind of attack, our third

assumption is that the module loader authenticates LKMs before loading them. Fourth, the source code of the kernel and all kernel extensions (e.g., device drivers) is available for the static analysis part of our defense. Finally, we assume that the guest VM can be booted into a known-good state (e.g., through a secure boot [34]) for the VMM to setup protection on the guest kernel and the KQH guards. After this initial setup, the guest VM is open to external events and may be placed under attack at any time.

4 A Unified Static Analysis Framework and Toolset

4.1 Overview

We build a unified static analysis framework (Figure 8) and a set of tools that can be used to derive security specifications for legitimate K-Queue requests. We assume that source code of the kernel is available for a static analysis, and such code is the only one that can be trusted. We note that although details of different K-Queues may vary, their specifications can be derived by a common set of analysis tasks. For example, the top-level legitimate K-Queue callback functions can be derived by a *points-to* analysis of the function pointer embedded in the respective K-Queue request data structures (e.g., Figure 2 – Figure 4); and every legitimate K-Queue callback function that takes a data parameter needs to go through a *transitive closure* analysis (Section 4.5.2). Therefore, we develop basic analysis tools and an analysis engine that composes these basic tools to carry out the analysis for each K-Queue instance.

Our analysis framework has the following advantages:

- General: the same framework engine can be used by any K-Queue instance, with only different starting *seed* analysis tasks (points-to analysis). Other than that, all K-Queue analysis proceeds in a similar fashion. This ensures that our framework can handle all types of K-Queue instances.

- Incremental: our framework uses a database to store basic analysis results, which enables accumulation of static analysis results over time, and more importantly, it facilitates sharing of basic analysis results among different K-Queue analyzers (Section 4.7).
- Automated: we develop a set of static analysis tools that can process the kernel source code and generate stubs of the corresponding guard code. Such automation greatly simplifies the job of a human analyzer.

4.2 Basic Analysis Tasks

One of the basic analysis tasks is points-to analysis [31] of function pointers, since our defense needs to know the legitimate targets of function pointers. For example, the top-level legitimate K-Queue callback functions can be recognized in this way. Simply put, they can be derived by analyzing which functions are assigned to the callback function field of K-Queue request structures. Table 1 summarizes possible ways that a callback function field can be assigned for different K-Queues.

- Direct assignment (DA). Here a callback function is directly assigned to the appropriate field.
- Indirect assignment (IA) through an intermediate expression (e.g., `do_floppy` in Table 1). This way is similar to the DA case except that an expression that points to a legitimate callback function is assigned to the callback function field. In Table 1, `do_floppy` may point to several possible functions under different conditions.
- Assignment through a function parameter (PA). This is a structured way of assignment in which a requester can call a wrapper function which in turn initializes a K-Queue data structure. The actual callback function or a pointer expression is passed in as a parameter to the wrapper function.

Accordingly, we can decompose the points-to analysis task into three kinds of simpler tasks: direct assignment analysis, indirect assignment analysis, and parameter assignment analysis.

Another basic analysis task is transitive closure analysis, which identifies constraints on the data parameters passed on to a legitimate target function. For example, the K-Queue instances all pass a requester-supplied data parameter to the callback function. If the callback function makes control transfer decisions based on the data parameter, we must make sure that the attacker cannot supply a malicious data parameter to induce kernel control flow to the malware's choosing (e.g., via a type 2 attack as in Figure 6).

4.3 The Analysis Engine

The heart of the K-Queue static analysis framework is the analysis engine, which repeatedly consumes individual analysis tasks (Section 4.2) from a work list. This work list is dynamically changing: on one hand, tasks are removed from it by the analysis engine; on the other hand, new tasks may be appended to it as a result of performing an analysis task. The analysis process is bootstrapped by some *seed* tasks inserted to the work list by a human analyzer, and it finishes when the work list becomes empty.

During each individual analysis task, the analysis engine runs one or more of the basic tools on the kernel source file as necessary and generates three kinds of output: (1) source code for the guard of a pending K-Queue request, (2) static analysis results that are stored to a database for reuse, and (3) detailed logs for in-depth diagnosis by a human analyzer.

4.4 The Work List

The work list contains pending static analysis tasks. Each element in this list specifies the type of analysis (direct assignment, parameter assignment, or transitive closure) and the corresponding input parameters. One example is <DA, tasklet_struct, func, 1>, which instructs

the analysis engine to invoke the direct assignment collector for the `func` field of kernel data structure `tasklet_struct`. Here "1" means that the first parameter of the callback function depends on the `data` attribute of the tasklet request. This task can bootstrap an analysis for the tasklet queue (Section 2), one of the K-Queues.

4.5 Basic Tools

These are the building-blocks of the static analyzer that carry out the basic analysis tasks discussed in Section 4.2.

4.5.1 Points-to Analyzers

The *direct assignment collector* takes as input the name of a structure type (e.g., `irqaction`) and the name of a field (e.g., `handler`) within that structure, and outputs kernel functions that can be assigned to such a field. It traverses each assignment statement (`lval = rval`) of the kernel. If `lval` ends with a field with the specified name, this field belongs to a structure with the specified name, and `rval` is an actual function, the tool collects `rval` as a legitimate function. If `rval` is not an actual function (e.g., it is a function pointer variable or a formal parameter), the tool invokes a pointer analysis algorithm called OLF (one level flow) [35] to find the possible targets of `rval`, and then collect such target functions as the legitimate functions.

The *parameter collector* collects target functions that are passed to a wrapper function as an actual parameter and later assigned to a function pointer (i.e., in the Parameter Assignment case in Table 1). It takes as input the name of a wrapper function and the index of the parameter of interest. It traverses the entire kernel searching for each invocation to the specified function, and collects the actual parameter at the specified index. If the actual parameter is a real function address, it is directly collected as a legitimate function. Otherwise, the tool first invokes the OLF

algorithm to find the possible targets of the actual parameter, and then collect such target functions as the legitimate functions.

The *indirect assignment collector* is covered by the Direct Assignment Collector and the Parameter Collector through the OLF algorithm.

4.5.2 Transitive Closure Analyzer

This tool is a major component of the toolset. It takes as input the name of a function and a list of its formal parameters that are *tainted*, i.e., these parameters can influence the control flow of the given function in a malicious way. This tool performs a flow-sensitive and intra-procedural transitive closure analysis, starting from the given function and descending into functions called by the given function and so on. It is flow-sensitive because it propagates taint to downstream functions through parameters. It is intra-procedural because only downstream functions defined within the same source file as the given function are analyzed. In case that a downstream function is located in a different source file, an external transitive closure analysis task is scheduled for execution later.

This tool builds a hash table of all functions defined in the given kernel source file, so that it can quickly navigate to any function to continue the analysis. It also maintains a list of functions that needs to be analyzed (called an *internal work list*). Initially, this list contains only the function given as the input of this tool. As this tool processes the given function, it may recognize more functions that need to be analyzed; then it adds such functions to the internal work list. The main body of this tool is a loop over the work list until it becomes empty. For each function in the list, this tool performs two kinds of tasks: taint propagation and new analysis task recognition.

- Taint propagation. The tool traverses each assignment statement (`lval = rval`) in the function and taints the variable `lval` if any part of `rval` is already tainted.
- New analysis task recognition. The tool traverses each function call statement `fn(args)` in the function to see if any part of `fn` or `args` or both is a tainted variable. If `fn` is tainted, a new points-to analysis task is generated for `fn` after the corresponding structure name and field name are derived from `fn`. If `args` is tainted and `fn` is an actual function, a transitive closure analysis task is generated for `fn` with the list of tainted arguments. If both `fn` and `args` are tainted, the analysis engine will generate new transitive analysis tasks for each target function of `fn`.

The work list maintained by the transitive closure analyzer is called an internal work list to differentiate it from the external work list (Section 4.4) used by the static analysis engine in Figure 8. New points-to analysis tasks are added to the external work list. New transitive closure analysis tasks are first added to the internal work list, and if they cannot be handled because the corresponding function is not defined within the given kernel source file, they are added to the external work list with the hope that they will be found in some other source file.

4.6 Kernel Merging

In order to speed up transitive closure analysis, we merge the entire kernel (given a configuration) into a single source file, so that the interprocedural analysis tasks are all turned into intraprocedural analysis. We test our analysis engine on a series of merged kernel source files in Section 6.2.

It is worth noting that depending on the configuration, a kernel compilation may include only a subset of all available source code, at least for the Linux kernel. For example, if a device driver is configured as a LKM (loadable kernel module), its code will not be included in the fixed part

of the kernel. This is not good for our K-Queue analysis because the kernel merging only includes source code of the fixed part of the kernel but we aim to cover all kernel source code including that of the device drivers. Therefore, we make a "total" configuration that has all device drivers built in the kernel, so that the kernel merging can reach maximum coverage of available kernel code. This is important for our support for LKMs.

4.7 Result Database

We introduce a database of individual points-to and transitive closure analysis results that is used to cache static analysis results and facilitate sharing among different K-Queue analyzers. This database prevents a K-Queue analyzer from performing redundant analysis tasks that have been done by another K-Queue analyzer, thus improves the overall performance. For example, sharing through this database decreases the execution time of the soft timer queue analyzer from 284 minutes to 127 minutes on a kernel of 482,369 lines of code.

Our database contains two tables: `pointsTo` and `transClosure`. When the analysis engine (Section 4.3) sees a points-to analysis task, it first uses the structure and field names as a key to query the `pointsTo` table. If a row is found, it directly uses the returned points-to set. Otherwise, it invokes the points-to analysis tools (e.g., the Direct Assignment Collector or the Parameter Collector) and inserts the results to the `pointsTo` table. The analysis engine uses the `transClosure` table in a similar fashion except that it uses the function name and the list of tainted arguments as search keys.

4.8 Code Generation for the KQH Guards

The analysis tool generates code stubs for the KQH guards. The generated code includes two kinds of functions: those for checking the control flow integrity of a function pointer and those for checking the control flow integrity of a real function. Figure 9 shows an example of the first

kind of function, and Figure 10 shows an example of the second kind. The code in Figure 9 is generated after the points-to analysis for structure `hwif_s` and field `ide_dma_test_irq` is finished, and the code in Figure 10 is generated when the transitive closure analysis for function `__ide_dma_test_irq` finishes.

The main body of the code in Figure 9 performs a series of comparisons to match the runtime value of a function pointer to a real function in its points-to set. If a match is found, the integrity of the function pointer is reduced to that of the matching real function. If no such match is found, the function pointer has no integrity because it points to something unexpected. In other words, the integrity of a function pointer is the disjunction (logical OR) of the integrity of all its legitimate targets (real functions).

Similarly, the integrity of a real function is the conjunction (logical AND) of the integrity of all function pointers that it transfers control to, and if no such function pointers are used, the function has integrity by default. For example, the code in Figure 10 can check the control flow integrity of `__ide_dma_test_irq` because the latter invokes one function pointer, which has structure name `hwif_s` and field name `INB`.

Note from Figure 9 that the function pointer checker uses kernel symbol names (e.g., `__ide_dma_test_irq`) to invoke `symbol2addr`, which tries to find the runtime address of a kernel symbol (including a function name). This use of runtime address lookup is necessary because of loadable kernel modules (LKMs) – if a device driver is configured as a LKM, the runtime address of its callback function cannot be predicted at static analysis time.

Also note from Figure 9 that before comparison the code fetches the runtime value of the function pointer from a pointer expression starting from data (e.g., `data->tx_timeout` as in

Figure 6). The determination of this pointer expression is not fully automatic yet in our current implementation.

5 Implementation

5.1 The K-Queue Analyzers

We implement the static analyzers for the IRQ action queue, the tasklet queue, the task queue, and the soft timer queue based on our static analysis framework, using CIL (C Intermediate Language) [36]. We implement the analysis engine in Shell scripts, which invokes our CIL modules that implement the basic analysis tools (Section 4.5). CIL provides an implementation of OLF [35], but it is not field-sensitive, so we improve it to satisfy our needs. All of our CIL modules are written in Objective Caml[1]. We use MySQL[2] (version 5.1.34) to store the result database (Section 4.7), and write a Java program to insert into or query the result database.

5.2 The K-Queue Guards

We implement four prototype guards for the IRQ action queue, the tasklet queue, the task queue, and the soft timer queue, based on the code stubs generated by the K-Queue analyzers (Section 4.8). They inspect the runtime status of the guest kernel to determine whether a pending K-Queue request is legitimate.

Next, we modify the servers of the IRQ action queue, the tasklet queue, the soft timer queue, and the task queue in the guest kernel, so that a guard is triggered before a pending K-Queue request is invoked. The modification to each K-Queue server is minimal, e.g., it changes `t->func(t->data)` in linux-2.4.32/kernel/softirq.c to

```
if (tasklet_guard(t->func,t->data) == true)
      t->func(t->data)
```

[1] http://caml.inria.fr/ocaml/index.en.html

[2] http://www.mysql.com

```
else printk("warning…");
```

5.3 Fine Grain Memory Protection

To write-protect the K-Queue guards and to defend against TOCTTOU attacks (Section 3.3), we need byte-level, fine grained memory protection (because a checked function pointer occupies only a few bytes in memory). However, the current architectures (e.g., Intel) can only support page-level write-protection; this is known as the protection granularity gap [18]. In order to overcome this gap, we are aware of two options: one is to build our defense on top of promising future architectures such as MemTracker [37] that supports fine-grained memory protection in hardware; the other is to use a software-based solution such as hook indirection in HookSafe [18]. In this paper, we implement a proof-of-concept memory protection scheme that does not overcome the protection granularity gap; our goal is mainly to demonstrate the effectiveness of our approach, and we leave the performance optimization as future work.

Specifically, we run the guest kernel on top of Xen 3.3.0 [32] VMM and extend the shadowing-based memory management of Xen for full virtualization to support fine grain memory protection. A new hypercall (`prot_range`) is added to allow the guest kernel to request regions in its address space to be write-protected. The size of the protected regions can be any number of bytes. The VMM sets the protection bit of a page table entry to read-only in the SPT (shadow page table) if that page contains any portion of a protected region. Then we modify the page fault handler of Xen so that it denies write attempts to the protected regions on this kind of page, but allows legitimate writes to other parts of this kind of page to go through.

During the K-Queue request checking in a guest VM, a KQH guard first write-protects each participating structure field (using a `VMCALL` instruction that invokes the `prot_range` hypercall) and then checks its value. When the check fails at any point, the already protected

structure fields are unlocked using another call to `prot_range`. If the check succeeds, the unlocking is deferred until the callback function has finished.

6 Evaluation of the K-Queue Defense

6.1 Effectiveness against KQH Attacks

We test the effectiveness of our K-Queue defense by running synthetic proof-of-concept rootkits in the guest kernel, including the key logger and the CPU cycle stealer described in [22], and the key logger described in Section 2. These rootkits leverage the Linux K-Queues in Table 1 and employ type 1 or type 2 attacks (Section 3.2.3). We observe that our K-Queue guards can immediately detect such rootkits. This shows that our defense is effective again KQH attacks.

Our K-Queue defense takes the following measures to reduce the likelihood of false negatives, i.e., a malicious K-Queue request can evade our guards. First, we statically insert the KQH guards into the guest kernel and directly modify the kernel source code so that whenever a pending K-Queue request is about to be served, the corresponding KQH guard is invoked. Malware in the guest may try to modify the guest kernel's code at runtime to disable or bypass our KQH guards, but this will be defeated because the inserted code is protected from tampering by the VMM (Section 3.3). We assume that the modified guest kernel image (with the KQH guards built in) has integrity and the guest Linux goes through a secure boot phase [34] in which the integrity of the inserted code is verified (along with the rest of the kernel) and protection is placed on the inserted code before the guest kernel is open to external events including malicious attacks. Second, all possible function pointers occurring in the control flow of the callback function are checked no matter along which path they occur. Specifically, the transitive closure analyzer searches through every possible execution path (starting from the callback function) and recognizes all function pointers along the way. Some of the function pointers may not be called

in a particular invocation, but the analyzer conservatively reports all such function pointers for points-to analysis. Third, our memory region protection (Section 5.3) guarantees that the attacker cannot replace a legitimate function pointer after it is checked but before it is followed. Fourth, our current implementation of the indirect assignment (IA) is not precise due to the inherent limitations of points-to analysis [38]. Concretely, the OLF algorithm [35] is conservative which means that it may return a superset of the legitimate targets of a function pointer. Fortunately, the IA case is relatively rare in the Linux kernel. For example, out of 55 points-to analysis tasks for the task queue, only two require indirect assignments (IA) analysis, and we manually confirmed that the results from OLF are accurate. In the worst case, the "extra" target functions calculated by OLF are still a restricted set of legitimate kernel functions instead of malicious functions of the rootkits, and if the rootkit wants to reuse a legitimate kernel function for a malicious purpose, it has to choose from this restricted set, which significantly reduces the rootkit's chance of finding a suitable one.

Our implementation of the K-Queue defense also tries to reduce the likelihood of false positives, in which it raises alarms at legitimate K-Queue requests. One source of such false alarms is legitimate functions in device drivers built as LKMs. In order to cover LKMs, our static analysis part uses a "total" kernel configuration that includes all possible device drivers (Section 4.6), which ensures that the generated KQH guards are aware of the existence of such legitimate functions even if they are compiled into LKMs and loaded into the memory at runtime. Another source of possible false positives is our assumption that when a pending K-Queue request is checked, all function pointers that may be invoked are already initialized and will not change during the execution of the callback function; if this assumption is invalid, e.g., a relevant function pointer has not been initialized at the time of check (it will be initialized during the

execution of the callback function), a KQH guard may generate false alarms. During our testing and performance evaluation, we have not seen any such false positives.

6.2 Performance and Scalability of the K-Queue Static Analyzer

We test the performance of our K-Queue static analyzer on a series of 10 configurations of Linux kernel 2.4.32 with increasing complexity. The first configuration is a minimal kernel that can boot the guest virtual machine. It contains 482,369 lines of code, with essential support for IDE disk, ext3 file system, and TCP/IP networking. Each successive configuration includes more device drivers. The most complex kernel configuration contains 1,010,196 lines of code.

Each experimental run covers four kinds of K-Queues in the following order: task queue, tasklet queue, IRQ action queue, and soft timer queue. Initially the analysis result database is empty. As the analysis proceeds the analysis results are accumulated in the database. Each K-Queue instance takes advantage of analysis that has finished, including its own analysis tasks and the K-Queue instance(s) ahead of it. For example, the analysis for the soft timer queue uses some results of the IRQ action queue, so it takes less time than if it has no existing results to use.

The experimental run for each kernel configuration proceeds as follows. Each K-Queue analysis starts with a points-to analysis task. When the points-to set is determined, a round of transitive closure analysis is performed, one for each function in the points-to set. As the result of the transitive closure analysis, new points-to analysis tasks may be recognized. If this is the case, another round of points-to analysis is performed, which may lead to one more round of transitive closure analysis. This iterative process continues until the last round of transitive closure analysis recognizes no new points-to analysis tasks.

All the experiments run on a 3.0 GHz Intel Pentium 4 with 1 GB of RAM.

The first thing that we measure is the execution time of the K-Queue analyzers. Figure 11 shows the cumulative execution time at four milestones for different kernel configurations. For example, the curve marked as "IRQ action" represents the total analysis time for the task queue, the tasklet queue, and the IRQ action queue. The X-axis is the complexity of the kernel configurations measured in KLOC or "thousand lines of code", and the Y-axis is the cumulative execution time in minutes. The ten points on each curve correspond to the measurements for the ten kernel configurations, the left-most point corresponds to configuration 1, and the right-most point corresponds to configuration 10.

From Figure 11 we can see that in general the analysis time increases as the complexity of the kernel increases. However, it seems that the execution time is not a simple function of the kernel size. In fact, we can see flat segments as well as steep slopes on the curves, suggesting a non-uniform distribution of the K-Queue requesters in the kernel. For example, the first steep slope occurs on the IRQ action queue curve from configuration 2 to configuration 3. This is because configuration 3 requires more analysis tasks. For example, from configuration 2 to configuration 3, the points-to analysis for structure `hwif_s` and field `ide_dma_test_irq` returns six more actual functions. These functions belong to the device drivers for several kinds of IDE controller chipsets (including the CMD64 series of chipsets and the HPT36X/37X chipset) that are added in configuration 3. It is these new actual functions that demand more transitive closure analysis than configuration 2. However, from configuration 3 to configuration 4 the IRQ action queue curve is pretty flat, because there are few new analysis tasks.

The way that the execution time curves look like is expected, because our choice for new kernel configurations is agnostic to K-Queue usage.

Figure 12 shows the number of external transitive closure analysis for the four kinds of K-Queues and different kernel configurations. Since we use merged kernel source files, all such analysis is due to new results from points-to analysis. Clearly, this number increases as the kernel size increases. The reasoning is as follows. As the size of the kernel grows, more source code is analyzed; then the number of requesters for a particular K-Queue is potentially increased. This leads to a larger points-to set for the top level function pointers, thus more functions that need transitive closure analysis. The new transitive closure analysis may detect new points-to analysis tasks, which result in more transitive closure analysis, and so on.

The above reasoning is supported by Figure 13, in which we show the measurement of the number of points-to analysis during the experiments. We can see that for all four kinds of K-Queues, the number of points-to analysis tasks indeed increases with the size of the kernel.

Figure 14 shows the cumulative number of internal transitive closure analysis (Section 4.5.2) during the experiments. The curves have a similar trend as the number of external transitive closure analysis and points-to analysis, but at a much larger scale (20x). This demonstrates the benefit of kernel merging (Section 4.6): if it is not used, a large number of such internal transitive closure analyses would become external transitive closure analyses; then the total analysis time would increase dramatically. This is because an external transitive closure analysis is more time-consuming than an internal transitive closure analysis. Each external transitive closure analysis has a constant overhead of preprocessing and parsing the entire kernel source code, while internal transitive closure analysis does not incur such overhead. As the kernel becomes more complex, such overhead becomes more and more significant.

One interesting point in Figure 14 is that up until configuration 6 the soft timer queue accounts for the most internal transitive closure analysis among the four K-Queues. But starting

from configuration 7, this dominance is lost to the IRQ action queue, and the number of internal transitive analysis for the soft timer queue even drops from 4,457 in configuration 6 to 2,715 in configuration 7. This is a correct behavior, because the number of internal transitive closure analysis for the IRQ action queue increases dramatically from 3,449 in configuration 6 to 9,485 in configuration 7, in such a way that it covers a significant portion of the analysis for the soft timer queue. As a supporting evidence, the analysis for the IRQ action queue took 1,548 minutes in configuration 7, which is significantly longer than that for configuration 6 (630 minutes), as shown in Figure 11.

6.3 Performance Overhead of the K-Queue Guards

To measure the runtime overhead of our K-Queue Guards, we choose five synthetic workloads and compare their execution time in different environments. These five workloads are: *cat* - read and display the content of 8,000 small files (with size ranging from 5K to 7.5K bytes) in a complicated directory tree; *ccrypt* - encrypt a text stream of 64M bytes, where ccrypt[3] is an open source encryption and decryption tool; *gzip* - compress a text file of 64M bytes using the `--best` option; *cp* - recursively copy a Linux kernel source tree; and *make* - perform a full build of the Apache HTTP server (version 2.2.2) from source. These benchmarks run on a 2.5 GHz Intel Core 2 Duo with VT-x support, and the guest VM is allocated 256 MB of RAM. The hypervisor is Xen 3.3.0, and the guest kernel is Linux 2.4.32 with configuration 1 (Section 6.2). Each experiment is run 10 times and the mean and standard deviation of the measurements are computed. Table 2 shows the results.

Table 2 contains three kinds of results. The "Original" results are collected on unmodified Xen and guest kernel and serve as the baseline. The results marked as "K-Queue" are collected

[3] http://sourceforge.net/projects/ccrypt/

on the modified Xen and the modified guest kernel with the four K-Queue guards (Section 5.2), but with the page-level memory protection (Section 5.3) turned off. Finally, the results marked as "K-Queue-Mem-Prot" are collected on the full-fledged defense mechanism including the modified Xen, the modified guest kernel, and the page-level memory protection.

From Table 2, we can see that if we do not write-protect the checked function pointers, our implementation of the K-Queue Guards incurs performance overhead ranging from almost negligible for *ccrypt* to 10% slow down for *cp*. However, when we turn on the memory protection to write-protect checked function pointers during the execution of K-Queue callback functions, the performance overhead increases for each of our workloads. For example, the overhead of *cat* increases from 4.2% to 11.3%. The most dramatic increase happens for *cp*, which jumps from 10% to 15 times slowdown. This is not very surprising given the fact that our proof-of-concept implementation leverages the page-level protection support of the hardware to achieve byte-level memory protection – if a memory page is made read-only because it contains protected memory regions, normal writes to other places on that page will not go through without triggering a page fault, and a large number of page faults can slow down the system significantly. However, from Table 2 we also see that the performance overhead is not always unacceptable even if we use page-level memory protection. Instead, it depends on the workload (e.g., it is 11.2% for the *gzip* benchmark).

In order to understand the result better, we carry out an event analysis of the K-Queue runtime defense. We found that the slowdown factor for each workload (see Table 2) depends on the complexity of the workload in terms of (1) what kinds of K-Queue callback events it triggers, and (2) how frequently it triggers such events. Some K-Queue callback functions are very complicated, so they require a large number of function pointer checks. For example,

ide_intr (linux-2.4.32/drivers/ide/ide-io.c), the IRQ action callback function for the IDE disk, requires a total of 192 function pointers to be checked; since these 192 function pointers may be scattered on many memory pages, they can affect a large number of normal writes to those pages and thus cause significant slowdown. The *cp* benchmark has the highest performance penalty because *cp* demands frequent disk write operations and accordingly frequent callbacks to ide_intr (42 times per second in our evaluation), and we know that the check of ide_intr is very complicated.

To further understand the performance overhead, we define and measure two complexity metrics of the K-Queue guards: layer and fanout. First we give an informal definition of the *layer* of checking: each layer is associated with a function pointer. The checker starts in layer 1, where the associated function pointer is the top-level K-Queue function pointers embedded in the K-Queue data structures. At layer i the value of the function pointer is first checked against a white list; if the check is successful then the integrity of the target function itself needs to be checked, which may require the checking of a new function pointer. In this case the check enters a new layer i+1. When the checking for a target function completes, the checker returns to the previous layer (i.e., layer i). We also define the *fanout* of a function as the number of function pointers whose integrity needs to be checked for that function.

For our K-Queue guards, the maximum layer during the checking of the IRQ action queue is seven, which happens when the top-level callback function is ide_intr. And during the checking of the IRQ action queue, the maximum fanout is 15 (for idedisk_error in linux-2.4.32/drivers/ide/ide-disk.c).

6.4 Discussions and Future Work

We are aware of several limitations of our current approach and proof-of-concept implementation. First, we assume that when a pending K-Queue request is checked, all function pointers that may be invoked are already initialized and will not change during the execution of the callback function. In other words, our current analysis does not handle indirect pointers or pointer fields that are written during the execution of the callback function. Second, our approach requires the source code be available for a static analysis, which means that users of closed source operating systems cannot employ our static analysis approach themselves. One related issue is the inclusion of new device drivers in the trusted kernel code base. The device vendor can perform the K-Queue static analysis and submit the generated guard code to the OS vendor which then integrates it into the final KQH guards. Third, in terms of the performance of our K-Queue analyzers, we see that they can become slow when the kernel is big. We have not focused on performance optimization so far because precision has been the most important for the correctness of our approach; since the static analysis happens offline, speed is less important. Fourth, our current implementation of K-Queue guards incurs significant overhead in some cases, mainly due to our implementation of the memory protection mechanism that employs page-level locking. One possible solution to reduce the overhead is to apply the hook indirection idea of HookSafe [18], in which we can relocate memory regions that need protection to a page-aligned memory space. Finally, whether our implementation has covered all possible KQH attacks in the Linux kernel is not proved yet, but our approach is general to any KQH attack. Specifically, our static analysis framework can work for any K-Queue given the correct seed task is provided (Section 4.1). How to automatically recognize all K-Queues that can be leveraged by KQH attacks is our ongoing research.

7 Related Work

Rootkits have attracted a lot of attention in the research community. Existing work can be classified into three areas: detection, defense, and analysis.

Related work in detecting rootkit execution includes integrity-based approaches such as Gibraltar [1], HookScout [3], Livewire [4], Copilot [6], SBCFI [7], Patagonix [15], and System Virginity Verifier [8], and cross-view based approaches such as Strider GhostBuster [9] and VMwatcher [5]. However, such techniques cannot detect KQH attacks. For example, SBCFI [7] is a checker for persistent kernel control flow attacks. It runs periodically to perform a garbage-collection style traversal of kernel data structures to verify that all of the function pointers target trusted addresses in the kernel. SBCFI can potentially catch a type 1 malicious K-Queue request, but cannot detect type 2 requests because it does not follow the generic data field (e.g., `void * data` in Figure 4) included as part of the request. In order to make SBCFI work on KQH attacks, accurate type information for the data field in each callback request must be added, which would require a static analysis of all K-Queue callback functions. Moreover, SBCFI's periodic checking is vulnerable to a Time-Of-Check-To-Time-Of-Use race condition [29]

KOP [2] is the closest to our KQH guards in terms of recognizing all function pointers that can be invoked by a K-Queue callback function. However, it cannot prevent a malicious K-Queue request from running since it is designed as a *passive* detector. Our KQH guards can deny the execution of malicious K-Queue requests.

In terms of rootkit prevention, related work includes CFI [28], Program shepherding [39], SecVisor [17], NICKLE [16], and HookSafe [18]. SecVisor and NICKLE are designed to preserve kernel code integrity or block the execution of foreign code in the kernel, but KQH attacks modify kernel data (add entries to queues) and can reuse existing kernel code, so they

cannot completely prevent KQH attacks. HookSafe cannot prevent KQH attacks because KQH attacks do not modify existing kernel hooks but supply their own kernel hooks, while HookSafe only protects persistent hooks. A more general approach, CFI [28] uses inlined reference monitors to preserve control flow integrity of programs, including control transfers through function pointers. Therefore, theoretically CFI can be instantiated into an alternative implementation of our K-Queue guards, and CFI can cover more than just the function pointers occurring in K-Queue callback function executions. For example, previous research finds that the number of function pointers can be thousands in a running kernel [3, 18], but the number of function pointers reachable from the four K-Queues that we study is only 487 (Figure 13), given the basic kernel configuration that we use. There are some difference between CFI and PLCP: CFI checks the integrity of function pointers individually and is unaware of the execution context; while PLCP checks function pointers in a batch based on a common context, i.e., the execution of a K-Queue callback function, which can potentially enable more precise, context-aware checks. Similarly, SBCFI performs checks without considering execution context. Program shepherding [39] prevents execution of injected or modified code in a single *user-level application* and relies on sandboxing the application, while KQH rootkits run at the kernel-level so program shepherding cannot be directly applied to KQH rootkits. In our previous work [22], we proposed a defense against *soft timer driven attacks* that are a subset of KQH attack. The limitations of [22] are coverage and scalability – it can only address soft timer based attacks, the static analysis tool is designed for the soft timer queue only, and the runtime checker is manually written; this paper presents a more general solution that can scale to any K-Queue instance. Moreover, this paper presents a new guarding architecture that significantly reduces the guarding overhead. In [22], we ran the guards in a special security VM and relied on cross-VM

introspection to query the status of the protected kernel running in a *different* guest VM than the security VM. In our evaluation with the soft timer queue only, the performance overhead was not an issue [22], but when we extended the guarding to the four K-Queue instances discussed in this paper, we observed unacceptable overhead (30x slowdown) in some cases. Therefore, in this paper we run the K-Queue guards in the same VM as the guest kernel under protection, in order to eliminate the cross-VM introspection cost. As a result, we reduce the overhead.

Finally, related work on rootkit analysis includes HookFinder [13], HookMap [12], Panorama [14], K-Tracer [10], and PoKeR [11]. HookFinder [13] proposes a fine-grained impact analysis to detect malware hooking behaviors, by identifying all the modifications made by the malicious code to its execution environment and keeping track of the impacts flowing across the whole system. HookFinder has the drawback that the implanted hooks by the malware may not be triggered when tested; and it can be evaded by malware applying "return-to-libc" techniques. Our KQH defense does not suffer from HookFinder's drawbacks because the K-Queue analyzers explore every execution path of the callback function and can address "return-to-libc" attacks by performing transitive closure analysis.

8 Conclusion

We present a solution to kernel queue hooking (KQH) attacks that manipulate K-Queues to achieve stealthy and continual malicious function execution. Such attacks are actively used by advanced malware such as the Rustock spam bot, but they remain invisible to state-of-the-art kernel integrity monitors. We propose the PLCP (Precise Lookahead Checking of function Pointers) approach that checks the legitimacy of pending K-Queue requests by proactively checking function pointers that may be invoked by the callback function. To facilitate the derivation of function pointers and their legitimate target, we build a unified static analysis

framework and toolset that can generate specifications of legitimate K-Queue requests and turn them into checking code. Based on the automatically generated code, we build a proof-of-concept runtime reference monitor that intercepts K-Queue requests and checks their legitimacy. We test our ideas on four K-Queues in Linux and perform a comprehensive experimental evaluation of the scalability of our static analysis framework and toolset, which shows that different K-Queue analyzers have significant overlapping that can be exploited for better efficiency; and runtime evaluation shows that our K-Queue defense can successfully stop synthetic KQH attacks but it has high overhead that needs to be reduced before it can be widely deployed.

9 References

[1] Baliga, A., Ganapathy, V., and Iftode, L. "Automatic Inference and Enforcement of Kernel Data Structure Invariants." Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08).

[2] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. "Mapping Kernel Objects to Enable Systematic Integrity Checking." Proceedings of ACM Conference on Computer and Communications Security (CCS'09).

[3] Yin, H., Poosankam, P., Hanna, S., and Song, D. "HookScout: Proactive binary-centric hook detection." Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Bonn, Germany, July 2010.

[4] Garfinkel, T. and Rosenblum, M. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS '03).

[5] Jiang, X., Wang, X., and Xu, D. "Stealthy malware detection through VMM-based "Out-Of-the-Box" semantic view reconstruction." Proceedings of ACM Conference on Computer and Communications Security (CCS'07).

[6] Petroni, N., Fraser, T., Molina, J., and Arbaugh, W.A. "Copilot—a coprocessor-based kernel runtime integrity monitor." Proceedings of USENIX Security Symposium'04, August 2004.

[7] Petroni, N. and Hicks, M. "Automated detection of persistent kernel control-flow attacks." Proceedings of ACM Conference on Computer and Communications Security (CCS'07).

[8] Rutkowska, J. "System Virginity Verifier." Hack In The Box Security Conference, September 2005. Invisible Things website. http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt, accessed March 2011.

[9] Wang, Y.-M., Beck, D., Vho, B., Roussev, R., and Verbowski, C. "Detecting stealth software with Strider GhostBuster." Proceedings of the 35th International Conference on Dependable Systems and Networks (DSN '05), 2005.

[10] Lanzi, A., Sharif, M., and Lee, W. "K-Tracer: A system for extracting kernel malware behavior." Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09).

[11] Riley, R., Jiang, X., and Xu, D. "Multi-aspect profiling of kernel rootkit behavior." EuroSys '09: Proceedings of the 4th European Conference on Computer Systems, 2009.

[12] Wang, Z., Jiang, X., Cui, W., Wang, X. "Countering persistent kernel rootkits through systematic hook discovery." Proceedings of the 11th International Symposium On Recent Advances in Intrusion Detection (RAID'08).

[13] Yin, H., Liang, Z., and Song. D. "HookFinder: Identifying and understanding malware hooking behaviors." Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08).

[14] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. "Panorama: capturing system-wide information flow for malware detection and analysis." Proceedings of ACM Conference on Computer and Communications Security (CCS '07).

[15] Litty, L., Lagar-Cavilla, H. A., and Lie, D. "Hypervisor support for identifying covertly executing binaries." Proceedings of the 17th USENIX Security Symposium, 2008.

[16] Riley, R., Jiang, X., and Xu, D. "Guest-transparent prevention of kernel rootkits with VMM-Based memory shadowing." Proceedings of the 11th International Symposium On Recent Advances in Intrusion Detection (RAID'08).

[17] Seshadri, A., Luk, M., Qu, N., and Perrig, A. "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes." Proceedings of ACM Symposium on Operating Systems Principles (SOSP), 2007.

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

[19] Solar Designer. "Bugtraq: Getting around non-executable stack (and fix)." website. http://seclists.org/bugtraq/1997/Aug/63, accessed March 2011.

[20] Shacham, H. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)." Proceedings of ACM Conference on Computer and Communications Security (CCS'07).

[21] Hund, R., Holz, T., and Freiling, F. C. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." Proceedings of the 18th USENIX Security Symposium, 2009.

[22] Wei, J., Payne, B. D., Giffin, J., and Pu, C. "Soft-timer driven transient kernel control flow attacks and defense." Proceedings of the 24th Annual Computer Security Applications Conference, December, 2008.

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

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

[25] Decker, A., Sancho, D., Kharouni, L., Goncharov, M., and McArdle, R. "Pushdo/Cutwail: A Study Of The Pushdo/Cutwail Botnet." Trend Micro Technical Report, May 2009.

[26] Boldewin, F. “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.

[27] Prakash, C. “What makes the Rustocks tick!” Proceedings of the 11th Association of anti-Virus Asia Researchers International Conference (AVAR’08), New Delhi, India. http://www.sunbeltsecurity.com/dl/WhatMakesRustocksTick.pdf

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

[29] Bishop, M. and Dilger, M. “Checking for Race Conditions in File Accesses.” Computing Systems 9 (Spring 1996):131–152.

[30] Becher, M., Dornseif, M., and Klein., C.N. “FireWire: all your memory are belong to us.” Proceedings of CanSecWest, 2005.

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

[32] Barham, P., Dragovic, B., Fraser, K., et al. “Xen and the art of virtualization.” Proceedings of ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, Oct. 2003.

[33] Wei, J. and Pu, C. “Multiprocessors May Reduce System Dependability under File-based Race Condition Attacks.” Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), Edinburgh, UK, June 25 - 28, 2007.

[34] Arbaugh, W. A., Farber, D. J., and Smith, J. M. “A secure and reliable bootstrap architecture.” Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA, May 1997.

[35] Das, M. “Unification-based pointer analysis with directional assignments.” Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 35-46.

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

[37] Venkataramani, G., Roemer, B., Solihin, Y. and Prvulovic, M. “MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging.” Proceedings of the 13th IEEE International Symposium on High-Performance Computer Architecture (HPCA-13), pages 273-284, February 2007.

[38] Hind, M. “Pointer analysis: haven't we solved this problem yet?” Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 54-61.

[39] Kiriansky, V., Bruening, D., and Amarasinghe, S. “Secure Execution Via Program Shepherding.” Proceedings of the 11th USENIX Security Symposeum, August 2002.

Vitae

Jinpeng Wei received a PhD in Computer Science from Georgia Institute of Technology, Atlanta, GA in 2009. He is currently an assistant professor at the School of Computing and Information Sciences, Florida International University, Miami, FL. His research interests include malware detection, information flow security in distributed systems, could computing security, and file-based race condition vulnerabilities. He is a member of the IEEE and the ACM.

Calton Pu received the PhD degree from the University of Washington in 1986. He is a professor and the John P. Imlay Jr. chair in software at Georgia Institute of Technology, Atlanta, GA. He has published more than 60 journal papers and book chapters, 170 refereed workshop and conference papers in operating systems, transaction processing, systems reliability and security, and Internet data management. He has served on more than 100 program committees for more than 50 international conferences and workshops. He is a member of the ACM, a senior member of the IEEE, and a fellow of the AAAS.

Figures

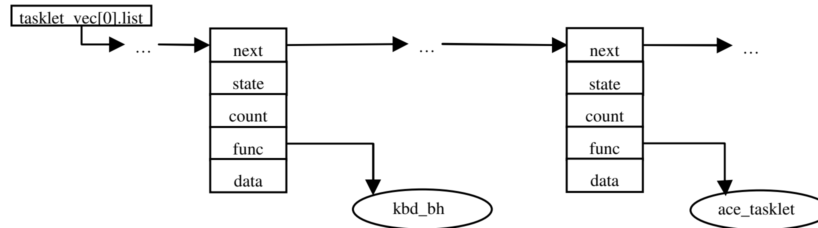

Figure 1: Illustration of the Tasklet Queue in Linux Kernel 2.4.32

<table>
<tr>
<td>struct irqaction {
  irqreturn_t (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};

Figure 2: The Definition of irqaction in Linux</td>
<td>struct tasklet_struct
{
  struct tasklet_struct *next;
  unsigned long state;
  atomic_t count;
  void (*func)(unsigned long);
  unsigned long data;
};

Figure 3: The Definition of tasklet_struct in Linux</td>
<td>struct tq_struct {
  struct list_head list;
  unsigned long sync;
  void(*routine)(void *);
  void *data;
};

Figure 4: The Definition of tq_struct in Linux</td>
</tr>
</table>

Linux:

IRQ action queue, tasklet queue, soft timer queue, task queue.

Windows:

IO completion routines, APC (Asynchronous Procedure Call) queues, threads saved context, protocol characteristics structure, driver object callback pointers, object deletion callback pointers, timers, DPC (Deferred Procedure Call) kernel objects, the IP filter driver hook, exception handler callback functions, TLS (Thread Local Storage) callback routines, plug and play notifications, process creation notifications, file system registration notifications, load image notifications.

Figure 5: Example Kernel Objects that Can Be Hooked

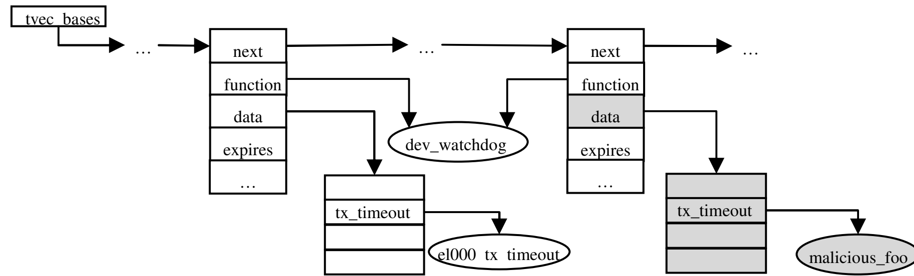

Figure 6: Illustration of a malicious soft timer request with a legitimate callback function (dev_watchdog in Linux kernel 2.6.16) and a malicious data pointer (shaded area means malicious). Here dev_watchdog may invoke a function pointer derived from the data field of the request [22]

Input: <*sn, fn*, *cb, d*>, where *cb* is a callback function stored in the *fn* field in a structure of type *sn*, and *d* is the data attribute.

Output: whether it is safe to invoke *cb* with *d* as an input parameter.

If *not* pointsTo(*sn*,*fn*,*cb*) return false;

Otherwise, return safeToExec(*cb*,*d*).

Here, $\text{pointsTo}(sn, fn, cb) = cb \in \text{P2Set}(sn, fn)$, and

safeToExec(*cb*,*d*) = $\bigcap_{s,f}(\text{pointsTo}(s, f, d') \text{ and } \text{safeToExec}(d', d))$, where *s* and *f* represent the type of a function pointer called by *cb* whose value is influenced by *d*, and *d'* is the actual value of that function pointer. If no such function pointer is invoked by *cb*, then safeToExec(*cb*,*d*) = *true*. The derivation of the expression is through a transitive closure analysis (Section 4.5.2).

P2Set(*sn*, *fn*) is the points-to set of the function pointer embedded in structure *sn* and field *fn*.

Figure 7: The PLCP (Precise Lookahead Checking of Pointers) Algorithm

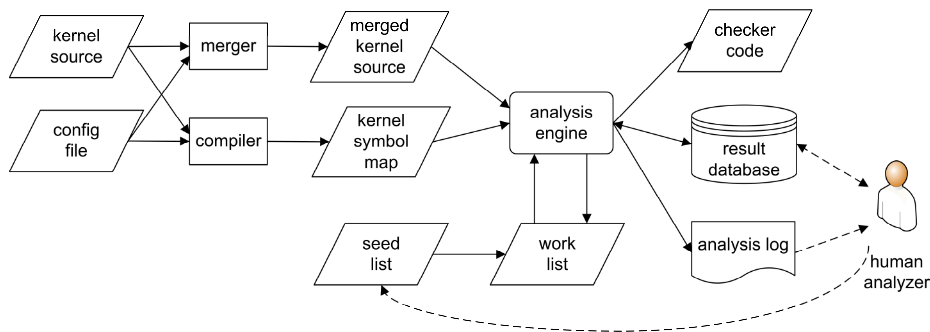

Figure 8: K-Queue Static Analysis Framework

```
int check_function_pointer_hwif_s_2_ide_dma_test_irq_1 (unsigned int data){
  unsigned int fp;

  /* Fetch the function pointer value into fp */
  /* fp = data-> … */
  if (fp == 0) return 1;

  if (fp == symbol2addr("__ide_dma_test_irq") )
    return check_function___ide_dma_test_irq_1(data);
  …
  unlock_kqueue_regions();
  return 0;  }
```

Figure 9: Generated Function Pointer Checker Code for Structure hwif_s and Field ide_dma_test_irq

```
int check_function___ide_dma_test_irq_1(unsigned int data){

 return 1
        && check_function_pointer_hwif_s_2_INB_1(data)
 ;
}
```

Figure 10: Generated Checker Code for the Real Function __ide_dma_test_irq

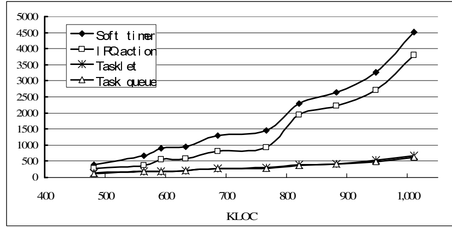

Figure 11: Cumulative Analysis Time (in minutes)

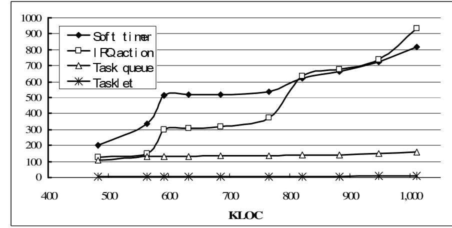

Figure 12: Number of External Transitive Closure Analysis

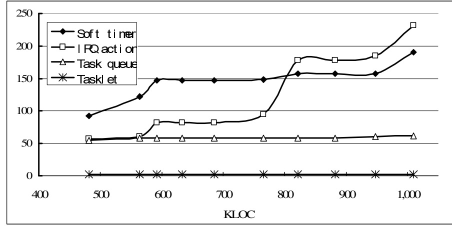

Figure 13: Number of Points-to Analysis

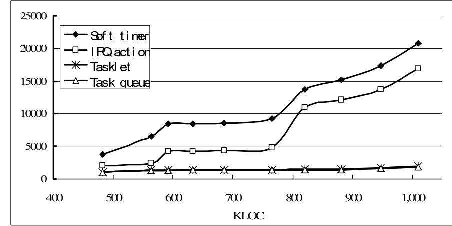

Figure 14: Number of Cumulative Internal Transitive Closure Analysis

Tables

Table 1: Possible Ways that a Callback Function can be Assigned in Different K-Queues

K-Queue Name	Structure Name	Field Name	Parameter Assignment		Indirect Assignment
			Function	Index (from 0)	
Soft timer queue	timer_list	function			
IRQ action queue	irqaction	handler	request_irq	1	
Tasklet queue	tasklet_struct	func	tasklet_init	1	
Task queue	tq_struct	routine	schedule_bh	0	do_floppy

Table 2: Overhead of the K-Queue Checker

	cat	**ccrypt**	**gzip**	**cp**	**make**
Original	15.03 ±0.38	3.10 ±0.03	5.79 ±0.05	45.61 ±4.62	143.29 ±3.57
K-Queue	15.66 ±1.15	3.13 ±0.02	5.97 ±0.22	50.19 ±5.67	148.31 ±3.63
Overhead	4.2%	1.0%	3.1%	10.0%	3.5%
K-Queue-Mem-Prot	16.73 ±0.93	3.58 ±0.03	6.44 ±0.35	747.80 ±21.44	187.74 ±19.82
Overhead	11.3%	15.5%	11.2%	1539.6%	31.0%