Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense

Jinpeng Wei, Bryan D. Payne, Jonathon Giffin, Calton Pu
School of Computer Science, Georgia Institute of Technology
{weijp, bdpayne, giffin, calton}@cc.gatech.edu

Abstract

A new class of stealthy kernel-level malware, called transient kernel control flow attacks, uses dynamic soft timers to achieve significant work while avoiding any persistent changes to kernel code or data. We demonstrate that soft timers can be used to implement attacks such as a stealthy key logger and a CPU cycle stealer. To defend against these attacks, we propose an approach based on static analysis of the entire kernel, which identifies and catalogs all legitimate soft timer interrupt requests (STIR) in a database. At runtime, a reference monitor in a trusted virtual machine compares each STIR with the database, only allowing the execution of known good STIRs. Our defensive technique has no false negatives because it mediates every STIR execution and prevents execution of all unknown, illegitimate STIRs, and no false positives because the relevant kernel code analyzed was unambiguous. The overhead for this additional security is less than 7% for each of our benchmarks.

1. Introduction

Internet-scale attacks, such as botnets, often utilize malicious software (malware) to hide their presence and extract information from their host systems. Rootkits, for example, are a common type of kernel-level malware that intercept and modify system events with the goal of hiding illicit activity [5, 12]. Other kernel-level malware can collect sensitive data, cause a denial of service, or open backdoors into the system. In this paper we present an attack technique that allows an attacker to execute kernel-level malware while evading detection from existing defensive tools. We then focus on techniques for detecting and mitigating the attack.

Attacks designed to maintain stealthy control of the kernel can be divided into three broad, and sometimes overlapping, categories: (1) detours attacks, (2) persistent kernel control flow attacks, and (3) transient attacks. The first category consists of malware that changes code on disk or in memory. These changes can be detected by trusted security tools that compare the current state of the system against a known good state. The second category consists of attacks that are capable of invoking malicious functions during execution by changing function pointers. The attacks in this category do not make any changes to the kernel code, but can be detected by control flow integrity (CFI) [1] and state-based control flow integrity (SBCFI) [20]. However, the third category of attacks is capable of evading current defensive techniques.

This category, *transient* kernel control flow attacks, is capable of achieving continual malicious function execution without persistently changing either kernel code or data. Transient attacks are also called soft timer attacks because they leverage the soft timer mechanism found in nearly all full-featured operating systems. These attacks are difficult to detect because many legitimate kernel components use soft timers and all soft timers share a *dynamic* queue, which prevents CFI and SBCFI from working in this scenario.

This paper has two primary contributions:

A concrete understanding of the severity of soft-timer attacks. We demonstrate that an attacker can use soft timer interrupt requests (STIRs) to perform powerful attacks including key logging, denial of service, and hidden process scheduling. We also show why current defensive tools do not work against these attacks.

A static analysis based tool that detects STIR attacks at runtime. We discuss the design, implementation, and evaluation of a new tool that detects STIR attacks. Under our security assumptions (Section 3.1), this tool detects all soft-timer attacks with less than 7% performance overhead.

The static analysis tool uses *summary signatures* to differentiate STIRs from legitimate and malicious software. Summary signatures characterize legitimate STIRs using callback functions and other constraints, and are derived through automated static analysis of the kernel source code. At run time, a reference monitor mediates STIR execution based on the summary signatures. We take several measures to protect the reference monitor, including executing it in a different virtual machine and using memory protections to prevent an attacker from bypassing the mediation step. Section 3 provides a complete discussion of our architecture and its security properties.

The rest of this paper is organized as follows. Section 2 provides background information on soft timers, and outlines three timer-driven attacks that we have developed. Section 3 presents our defense mechanism against such attacks. Section 4 described the Xen-based prototype imple-

1063-9527/08 $25.00 © 2008 IEEE
DOI 10.1109/ACSAC.2008.40

mentation of the defense and its evaluation in terms of effectiveness and performance overhead. Section 5 discusses related work, and we conclude in Section 6.

2. Soft Timer Based Attacks

Dynamic soft timers are a well-established mechanism used by many kernel components to schedule the execution of a timed event handling function [4]. Common uses of soft timers include retries when polling a physical device, retransmission of data, and handling of network protocol timeouts. Unlike hardware timer interrupts, soft timers are execution requests that need to be scheduled. A soft timer interrupt request (STIR) typically specifies when to execute, a callback function, and a data pointer to uninterpreted contextual information. This request is saved in a queue by the kernel. To execute a STIR, the kernel invokes the callback function and passes along the data pointer as a parameter.

The control flows due to STIR call back functions are injected into the main kernel control loop upon request. Under the assumption that everything in the kernel space is equally trusted, such transfers of control are acceptable. However, if one of the requesters is malicious, the soft timer mechanism can be turned into a reliable way of maintaining stealthy control. An attack can be divided into a sequence of STIRs and executed using successive timer callback functions.

For ease of presentation, we adopt a simple and informal model of kernel-level malware that executes useful work for a botnet owner or renter. Under this model, the malware's life cycle can be divided into three steps: (1) system penetration, (2) interpose on the kernel control flow, and (3) continually execute malicious functionality. Penetration methods (step 1) such as buffer overflows [9] are well known and omitted from this discussion. Previous persistent kernel control flow attacks (e.g., the rootkits listed in [20]) change kernel data structures (step 2) to force the kernel to branch/jump to malicious functionality (step 3). Like persistent attacks, our new transient attacks interpose on the kernel's control flow (step 2) at the time of the attack. However, unlike persistent kernel control flow attacks, which typically replace a permanent function pointer in the kernel, a transient kernel control flow attack simply installs a malicious STIR. In our demonstration, malicious functionality is implemented using a Linux loadable kernel module (LKM) that requests the first STIR in its initialization function. When the malicious LKM is loaded, the kernel invokes its initialization function, and step 2 is completed. The malware's persistent execution (step 3) is possible because each STIR can request the next STIR that references the callback function. For added stealth, the location of this callback function can change with each STIR execution.

To understand the effectiveness of transient kernel control flow attacks, this section outlines the design of three soft-timer based malware examples to show that they can perform a wide variety of malicious objectives. These examples are implemented as LKMs and run through the soft

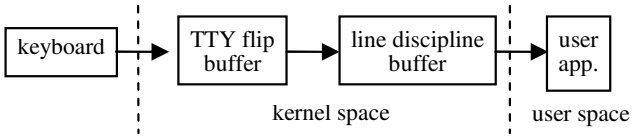

Figure 1: Flow of keyboard input in Linux.

timer facility. More specifically, they invoke the kernel API **add_timer** to request a STIR. **add_timer** takes as input a parameter that points to a data structure of type **timer_list**, and the *function* field of this structure is set to a callback function. A callback function is specific to the corresponding malware, but all such functions request the next STIR before they return, e.g., by calling **add_timer**.

The three malware examples below demonstrate violation of the three basic security properties: the stealthy key logger violates confidentiality, and the cycle stealer and the alter-scheduler violate both availability and integrity.

2.1. Stealthy Key Logger

A typical class of malware steals sensitive information from the host node. A straightforward but easily detected malware implementation intercepts the kernel functions that process such sensitive information. For example, a key logger [21] can replace the keyboard interrupt handler (e.g., IRQ 1) with a malicious handler that records the keyboard input. The following implemented example shows that persistent kernel modifications are not needed for this type of malicious functionality.

A timer-driven key logger keeps kernel code and interrupt-related data structures intact. It periodically looks at various buffers in the kernel, where the keyboard input information is stored. As Figure 1 shows, when a key is pressed, the keyboard hardware generates an interrupt. The keyboard interrupt handler fetches the key stroke information and temporarily stores it in the TTY flip buffer before transferring it into the TTY line discipline buffer. Finally, when a user-level application reads from the standard input device, keystroke data is copied into the user's buffer.

The sampling rate determines whether or not a timer-driven key logger can capture every keystroke. The key logger can obtain keystroke information from the TTY flip buffer, the TTY line discipline buffer, or the user's buffer. The TTY flip buffer has a very short retention time relative to the TTY line discipline buffer, which is a large circular buffer (normally 4096 bytes). Since each keystroke generates 2 bytes of information, the TTY line discipline buffer can keep information on up to 2048 keystrokes. Since it can take several minutes for the average user to fill up the line discipline buffer, the key logger malware only needs to inspect the buffer periodically (e.g., once per minute should be good enough) to collect all of the user's keystrokes. In the event that more frequent sampling is required, the key logger can request faster soft timer interrupts. In this case, techniques for hiding the higher resource consump-

tion should be employed (see Section 2.2) to keep the key logger stealthy.

We have implemented the sampling key logger on Linux to collect key strokes from an X Window desktop. It captures keystrokes entered into X Window applications, including the gedit editor, the Firefox web browser, and terminal window emulators. These applications handle many security-critical keystrokes including usernames, passwords, and credit card numbers.

2.2. Stealthy Denial of Service Attack

A second common type of attack causes a denial of service (DoS) or lowered quality of service. In a soft timer-driven attack, the call back function can perform computationally intensive work to steal system resources thereby slowing down or halting any legitimate application. One simple CPU cycle stealer has been implemented by inserting a program to compute the factorial of a given number in the call back function. By adjusting the value of the number and the timer's period, different slowdown factor can be obtained. We measure the CPU usage during such an attack where the timer's period is fixed at one second. When the value of the number is below 25, the CPU consumption by the malware is negligible. As the value becomes larger, there is an exponential increase in the CPU consumption by the malware. For example, when the value is 36, the CPU consumption is about 54%, and when the value grows to 42, the CPU consumption is close to 100%. Note that the actual algorithm used to steal CPU cycles is irrelevant to the attack. Instead, this attack shows that a resource exhaustion attack can be stealthily deployed, preventing the system from performing its intended tasks.

The attack becomes effective when the malware is able to hide itself and its effects from detection for a significant amount of time. One problem with typical DoS attacks is that the wasted CPU cycles are detectable by system tools such as *top*. This is because the kernel maintains performance accounting information for different sources of computation. For example, the CPU time consumed by the above malicious call back function is attributed to "software interrupt". To hide this attack, the malicious call back function further manipulates the kernel accounting data (e.g., kstat_cpu(0).cpustat) such that the CPU time used by the malicious STIR is attributed towards the idle CPU time. Therefore, it is not immediately obvious why the system performance is degrading.

Our CPU cycle stealer violates the availability of CPU resources and the integrity of the performance accounting information. However, since the performance accounting information is dynamic, there is no easy notion of what is normal. Under such attacks, a system may report slowdown of a service, but there can be many other reasons for poor performance (network congestion, server overload, retries due to device error, etc). Therefore, this type of attack is not easily discovered.

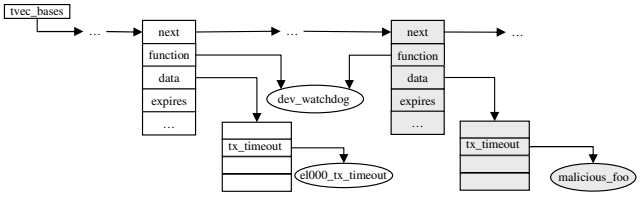

Figure 2: Illustration of a malicious STIR with a legitimate callback function (dev_watchdog in Linux kernel 2.6.16) and a malicious data pointer (Shaded area means malicious). Here dev_watchdog may invoke a function pointer derived from the data field of the STIR.

2.3. Running a Hidden Process: the Alter-Scheduler

A third kind of malware, called alter-scheduler, is capable of running a malicious process without relying on the legitimate kernel scheduler. Some existing malware can hide a malicious process by removing its entry from the all-task linked list of the kernel. However, this malware must leave the malicious task structure in the run queue in order for it to be scheduled. Therefore, a detection tool such as [19] that cross checks the all-task linked list and the run queue is able to detect the malicious process.

The alter-scheduler malware implements a special scheduler exclusively for the malicious process. It keeps a record of the malicious process structure and detaches it from both the all-task list and the standard run queue. Within the STIR call back function, the alter-scheduler preempts the currently running task, as if a higher-priority process has become runnable. Then it forces a context switch to the malicious process, as if the malicious process has been chosen as the new task to run. The standard scheduler is resumed when the malicious process surrenders the CPU.

This style of attack is very powerful because the malicious process is made independent of (and thus invisible to) the legitimate kernel scheduler and other relevant routines, and the malicious alter-scheduler instead supplies the missing functionality (e.g., giving the malicious process opportunities to run). Therefore, malware based on the alter-scheduler can remain stealthy against state-of-the-art detectors such as [19].

3. Soft Timer Attack Detection and Defense

Each attack in Section 2 must usurp kernel control flow in order to execute malicious code. Soft timers can be leveraged to do this in one of two ways: (Type 1) supply a malicious timer callback function, or (Type 2) supply a legitimate timer callback function but a *malicious data pointer* such that the control flow of the legitimate callback function is modified to invoke malicious functionality as a subroutine (similar to the "jump-to-libc" style attacks [26]). The latter option is possible because when a STIR callback function is invoked, a data pointer embedded in the STIR is passed as the input parameter. In some cases, the STIR callback func-

tion may derive a function pointer from this input, thereby allowing the data to alter the control flow.

3.1. Security Assumptions and Threat Model

Our defensive techniques against soft timer attacks are based on four standard security assumptions. First, since we use a virtualization-based architecture, we assume that the virtual machine manager (VMM) and the security virtual machine (VM) are part of the trusted computing base. This assumption is based on the idea that the VMM code base can be small, and therefore auditable, and the interface between the guest VM and the VMM can be narrow and protected. Our second assumption is that the legitimate kernel code in the guest VM's memory can not be tampered with by malicious code. In a production setting, this must be enforced by existing security tools such as Copilot [18] or SecVisor [24]. Third, we assume that the source code of the kernel and all kernel extensions are available for the static analysis portion of our tool. Note that closed source operating system vendors could perform the static analysis and make the results available to the end-users. For open source operating systems, the entire procedure can be performed by end-users. Lastly, in order to provide protections for this system, we require that the system can be booted into a known good state (i.e., secure boot [2]). We then perform a brief initialization phase to setup our defensive system and then the guest VM is open to outside events and may be placed under attack at any time.

Our threat model allows an attacker to install malicious code on this system running at the highest privilege level. The attacker is able to perform kernel-level attacks, but we assume that protections are in place to prevent tampering with kernel code as described above. Under this model, the attacker is powerful and able to run soft timer attacks unless our defensive system prevents them. This is a realistic threat model and no more constraining to an attacker than previous work in this space [20].

3.2. Legitimate STIR Identification

The basic idea of our proposed defense is to validate each STIR before its execution, thereby preventing the execution of malicious STIRs. Based on the "fail-safe defaults" principle [23], we use a white list of STIR *summary signatures* to distinguish legitimate STIRs from malicious ones. An unknown STIR that does not have a matching STIR summary signature is considered suspect and denied execution.

3.2.1 STIR Summary Signatures

Recall that a malicious STIR can induce kernel control flow in two ways: (1) supply a malicious timer callback function, or (2) supply a legitimate timer callback function but a malicious data pointer. In order to detect type 1 malicious STIRs, we only need to check their callback functions against a white list of legitimate timer callback functions. However, in order to detect type 2 malicious

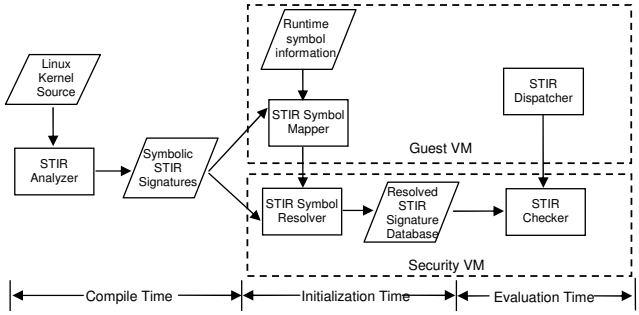

Figure 3: Processing STIR summary signatures.

STIRs, we must check the data pointer in addition to checking the callback function. Figure 2 illustrates a type 2 malicious STIR (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 STIR is set to a malicious function (e.g., malicious_foo). Therefore, we can detect this malicious STIR by comparing the tx_timeout field against a white list of legitimate functions (for example e1000_tx_timeout) that can be assigned to this field for the legitimate STIRs.

Consequently, we choose the STIR summary signature as a two-element tuple <*function*, *assertion*>, where *function* represents a legitimate timer callback function (e.g., dev_watchdog), and *assertion* represents properties of legitimate data passed to the legitimate callback function as input. Specifically, an assertion is the logical AND of 0 or more parameterized predicates. Each predicate has the form "*deref* equals *functionlist*", where *deref* specifies a way to dereference a function pointer (e.g., data->tx_timeout), and *functionlist* is the logical OR of one or more legitimate functions that can be assigned to the dereferenced function pointer. An example assertion associated with dev_watchdog is:

(data->tx_timeout **equals** (el000_tx_timeout OR xircom_tx_timeout))

Figure 3 shows the overall processing of the STIR summary signatures, divided into three phases corresponding to compile time, initialization time and evaluation time, respectively. In the first phase, Linux kernel source code is statically analyzed by the STIR Analyzer to generate the symbolic STIR summary signatures. These signatures are symbolic because the addresses of the functions in them may be unknown at compile time (e.g., due to dynamic kernel module loading). The actual mappings of these functions to their runtime addresses happen in the second phase (See details in Section 3.2.4), when the symbolic summary signatures become *resolved summary signatures*. This process is in some way similar to partial evaluation [8]. Finally, during the normal operation of the guest VM (e.g., the evaluation time), the STIR Checker (Section 3.3) uses the resolved summary signatures to prevent control transfers due to malicious STIRs.

In the first phase, the STIR Analyzer performs a top-level analysis (Section 3.2.2) to derive the *function* part of the STIR summary signatures and a transitive closure analysis (Section 3.2.3) to generate the *assertion* part of the STIR summary signatures. The latter analysis identifies all function pointer dereferences of the input parameter in the legitimate STIR callback functions, as well as all legitimate functions that they target.

3.2.2 Top-Level Analysis

We first consider the collection of legitimate STIR callback functions, which we call LegitTimerfuncs. These are the top-level functions that require validation before execution. Each function in LegitTimerfuncs will become the *function* part of a STIR summary signature after the transitive closure analysis (as described in Section 3.2.3).

t.function = fn;
t = TIMER_INITIALIZER (fn, expires, data);
DEFINE_TIMER(t, fn, expires, data);
setup_timer(&t, fn, data);

Table 1: Different ways of assigning timer callback functions in the Linux kernel

LegitTimerfuncs is constructed by scanning the kernel source code to identify all legitimate instances of soft timer callback functions. Table 1 shows the four techniques to link soft timer callback functions, denoted fn, to the timer_list structure, denoted t. The first is by assignment. The second and third techniques are macros that actually expand to assignment. Therefore the first three cases are analyzed in the same way: the STIR Analyzer traverses each assignment statement (lval = rval) of each function in the Linux kernel, and if lval ends with a field named function within a structure of type timer_list, then rval is recognized as a soft timer callback function. The last technique to link a soft timer callback function is to use the setup_timer procedure. This technique is handled by traversing each function call to setup_timer and collecting the second parameter in the function call.

We assume that benign programmers follow the standard APIs in Table 1 to request STIRs. Since the top-level analysis considers all 4 ways in Table 1, it can capture all legitimate STIR callback functions.

3.2.3 STIR Callback Transitive Closure Analysis

Verification of the top-level LegitTimerfuncs is insufficient to guarantee defense because it only addresses type 1 malicious STIRs and not type 2. To detect potential attacks in lower level subroutines, the second part of the STIR Analyzer checks the function calls within each callback function in LegitTimerfuncs to see if any of them allows indirect control transfers. Concretely, if function pointers are derived from the input parameter of a callback function that further branches to one of those pointers, then the analyzer raises a flag to indicate that the callback function needs a transitive closure analysis of all such pointers.

Transitive closure analysis of $fn(arg)$:
(1) Initially arg is added to $tainted\_vars$;
(2) For each assignment statement $lval = rval$ or $lval = f'(rval)$ in fn:
If any part of $rval$ is in $tainted\_vars$, then $lval$ is added to $tainted\_vars$.
(3) For each function call statement $f(params)$ in fn:
If any part of $f(params)$ is in $tainted\_vars$, then raise a flag for fn.

Figure 4: Analysis for STIR callback functions.

Figure 4 shows the high-level algorithm for the transitive closure analysis. Given a callback function fn with parameter arg, the STIR Analyzer first traverses each assignment statement of fn to compute the set of variables (tainted_vars) whose value can be influenced by arg, directly or indirectly. Next, the STIR Analyzer searches every function call statement of fn to see if the target function or its parameter is influenced by any variable in tainted_vars. Existence of such a function call means that control can go to places decided by arg, which could be exploited by malware.

If the STIR Analyzer does not raise a flag for a callback function in the transitive closure analysis, a signature <*function*, *assertion*> is completed where *function* is the name of the callback function, and *assertion* is simply the boolean value *true* (which means that no further check is needed on the data parameter arg of the callback function).

If the STIR Analyzer raises a flag, a further step is performed to compute the *assertion*. This step can be further subdivided into three cases.

Case 1: Only the function name part of a function call statement (i.e., f in f(params) of Figure 4) is influenced by the input parameter (arg), which means that arg is used to derive a function pointer. In this case, the third step decides the legitimate functions that can be assigned to the function pointer derived from arg. For each distinct way of dereferencing arg, a predicate "*deref* equals *functionlist*" (introduced in Section 3.2.1) is generated, where *deref* specifies the way to dereference arg, and *functionlist* is the logical OR of legitimate functions that can be assigned to the dereferenced function pointer. The *assertion* is the logical AND of all such predicates. The process of deriving legitimate functions in a predicate is similar to the top-level analysis (section 3.2.2) which identifies the timer callback functions.

Case 2: Only the parameter part of a function call statement (i.e., params in f(params) of Figure 4) is influenced by the input parameter arg. In this case, the same analysis in Figure 4 is applied to f, and all resultant predicates are appended (ANDed) to the *assertion*.

Case 3: Both the function name and the parameter of a function call statement are influenced by arg. The third step treats this case as a composition of case 1 and case 2.

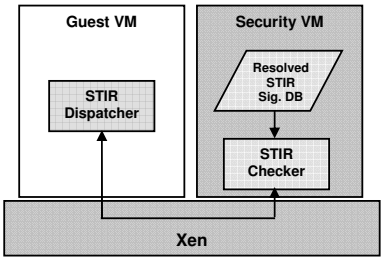

Figure 5: Defense against soft timer attacks.

i.e., it first processes the function name part to derive the legitimate functions and then processes the parameter part on each of the identified legitimate functions.

The STIR Analyzer relies on accurate type information to recognize function pointer dereferences. In the Linux kernel (written in C), addresses could be calculated by pointer arithmetic operations. In practice, we have found no such unsafe pointer arithmetic operations in all of the STIR related kernel functions we have inspected. Due to the threat represented by kernel control flow attacks (both persistent and transient), we encourage kernel developers to continue avoiding pointer arithmetic operations in legitimate kernel functions. This will help to support comprehensive kernel code analysis that depends on type information.

3.2.4 Generating Resolved STIR Summary Signatures

The outcome of the STIR Analyzer is the symbolic STIR summary signatures. These contain symbols (e.g., STIR callback function names) whose runtime *addresses* may not be determined statically. Specifically, Linux supports loadable kernel modules (LKMs) that can be added to the kernel at runtime. If a legitimate LKM uses a soft timer, the address of its callback function cannot be known until after the module is loaded. Therefore, we provide a mechanism to register such symbol-address mappings at runtime.

Because we employ a VMM-based detection architecture (described in Section 3.3), the registration mechanism is split into two components: a guest VM component (called a STIR Symbol Mapper) and a security VM component (called a STIR Symbol Resolver), as shown in Figure 3. At the guest VM initialization time, the STIR Symbol Mapper first generates mappings from function names in the symbolic STIR summary signatures to virtual addresses in the guest kernel's address space. It then informs the STIR Symbol Resolver about these mappings through an inter-VM communication. When the STIR Symbol Resolver receives the mapping list, it merges the addresses with the corresponding symbolic STIR summary signatures, which become resolved STIR summary signatures that can be used to check the legitimacy of pending STIRs.

3.3. The STIR Checker

Because soft timer attacks are at the kernel-level, a defense mechanism inside the same kernel would be vulnerable to tampering by an attacker. Consequently, an effective defense must be isolated from the victim kernel. Virtual machine managers (VMMs) are one environment that provides such isolation, allowing us to run the defensive mechanism in a VM that is isolated from the guest VM. Our implementation uses Xen [3] for these isolation properties.

As shown in Figure 5, our architecture places the STIR Checker outside of the victim guest kernel in a separate domain (called the security VM). The purpose of the STIR Checker is to prevent control transfers from the guest kernel to malicious functionality such as those outlined in Section 2. Specifically, the software timer dispatcher of the guest kernel is modified to inform the STIR Checker about the callback function and related *data* when a pending STIR expires, and invokes the callback function only if the STIR Checker returns *true*. During the time when the STIR Checker is making a decision, the guest kernel is suspended. The communication between the STIR Checker in the security VM and the guest VM is facilitated by an inter-VM communication channel. The modification to the guest kernel is protected using memory-protection capabilities from the Lares architecture [17]; therefore the STIR Checking cannot be trivially bypassed.

The STIR Checker module compares the next STIR to be dispatched against a list of resolved STIR summary signatures (Section 3.2.4). As Figure 5 shows, all STIR summary signatures are stored in a database, indexed by their *function* element (Section 3.2.1). Given a STIR, the STIR Checker first uses its function field as the index to look up the summary signature database. If a signature is found, and the *assertion* evaluates to *true* on the data field, the STIR is considered legitimate. Otherwise it is considered malicious.

4. Linux Implementation and Evaluation

4.1. STIR Analyzer

We used the Common Intermediate Language (CIL) [15] to implement a prototype STIR Analyzer, which consists of several program analysis modules that implement the algorithms in sections 3.2.2 and 3.2.3. These modules receive high-level representations of the kernel source files generated by CIL, analyze them, and output the results.

The STIR Analyzer can analyze the entire Linux kernel 2.6.16 in about one hour on our test system (a 2.4 GHz Intel Core 2 Duo with 2 GB of RAM). The analyzer found a total of 365 legitimate STIR callback functions in the 3688 kernel source files analyzed.

A majority of these STIR callback functions (333 out of 365) do not derive function pointers from the input parameter, therefore they can not be used to construct type 2 malicious STIRs (Section 3.2.1).

On the other hand, 32 of the 365 top-level callback functions do derive function pointers from their input parameter. Transitive closure analysis was carried out on these functions to identify the legitimate subroutines to which the derived function pointers can point. We describe these in

Source file	Timer Callback Function	Function Pointers Derived From Input
drivers/input/joystick/db9.c	db9_timer(struct db9 *private)	private->pd->port->ops->read_data private->pd->port->ops->read_status private->pd->port->ops->write_control
drivers/input/gameport/gameport.c	gameport_run_poll_handler (struct gameport *d)	d->poll_handler
drivers/isdn/hisax/isdnl3.c	l3ExpireTimer (struct L3Timer *t)	t->pc->st->lli.l4l3
drivers/scsi/scsi_debug.c	timer_intr_handler (unsigned long indx)	queued_arr[indx].done_funct
net/sched/sch_generic.c	dev_watchdog (struct net_device *arg)	arg->tx_timeout

Table 2: Representative STIR callback functions that need transitive closure analysis (Linux-2.6.16)

some detail, since they represent potential vulnerabilities (e.g., type 2 malicious STIRs).

Table 2 lists some of the 32 STIR callback functions that derive function pointers from the input parameter. From these functions, we can make the following observations. First, the dereferences in some functions are complicated. For example, the input parameter private in db9_timer goes through 4 layers of indirection before reaching a function pointer (private->pd->port->ops->read_data). Second, it is normal for a STIR callback function (such as db9_timer) to dereference the input parameter in multiple ways. Correspondingly the *assertion* part of the STIR summary signature for such a function will have multiple predicates (Section 3.2.1). Finally, most of the callback functions interpret the input parameter as a pointer to a structure. The only exception is timer_intr_handler in drivers/scsi/scsi_debug.c, which uses the input parameter as an index into a global array of structures. A function pointer is in turn retrieved from the array element indexed by the input parameter.

When a callback function such as dev_watchdog is encountered, the STIR Analyzer goes through a further step (the third step in 3.2.3) of transitive closure analysis. For dev_watchdog, the STIR Analyzer reveals 113 functions in the Linux kernel that can be assigned to dev->tx_timeout. Due to space limitations, only four are shown in Table 3.

Uses of the Symbolic STIR Summary Signatures. As shown in Figure 3, the output of the STIR Analyzer is the symbolic STIR summary signatures. We use this information to implement the rest of our defense. The usage falls into two categories: first, the function names in the symbolic summary signatures are retrieved and incorporated into the STIR Symbol Mapper (Section 4.2.1) in the guest kernel and the STIR Symbol Resolver (Section 4.2.2) in the security VM; second, the function pointer dereference information in the symbolic summary signatures are transformed into offsets within data structures (through an offline type analysis) and then incorporated into the STIR Checker (Section 4.2.3).

Function	Location
ace_watchdog	drivers/net/acenic.c
ariadne_tx_timeout	drivers/net/ariadne.c
arlan_tx_timeout	drivers/net/wireless/arlan-main.c
e1000_tx_timeout	drivers/net/e1000/e1000_main.c

Table 3: A sampling of legitimate functions that can be assigned to dev->tx_timeout in dev_watchdog

4.2. Implementation of the STIR Defense

Our implementation uses the Lares architecture [17] to transfer control to the STIR Checker in the security VM and to ensure that the STIR Dispatcher cannot be circumvented. Lares provides the infrastructure needed to place hooks into the guest kernel, which divert execution into the security VM. Lares also provides protections to ensure that these hooks are not tampered or circumvented.

This functionality is supported, in part, by a new hypercall (lares_op) that is effectively a system call from an operating system kernel into the VMM. The security VM first invokes lares_op to register a shared memory region for exchanging information between itself and the VMM. When the hook in the guest VM is triggered, a VMCALL to lares_op is made with input parameters that contain the hook's location and function arguments. Upon receiving the VMCALL, lares_op pauses the guest VM, copies the parameters from the guest VM to the memory region shared with the security VM, and triggers a virtual IRQ. The security VM handles the virtual IRQ by copying the event context from the guest into its address space. It then performs its monitoring function and places the response in the shared memory block. Next, lares_op is invoked again to inform the VMM that the response is ready. Upon receiving this hypercall, the VMM unpauses the guest and enforces the response from the security VM in the guest VM.

For this work, we extended Lares by defining a new parameter structure passed through the VMCALL from the guest kernel to the security VM. Two commands are defined in this structure: REGISTER_STIR_SYMBOLS, and CHECK_STIR. The first command is used by the STIR Symbol Mapper, and the second command is used by the modified soft timer dispatching logic.

4.2.1 Implementation of the STIR Symbol Mapper

The STIR Symbol Mapper (Section 3.2.4) is implemented in the guest VM as a loadable kernel module that notifies the STIR Symbol Resolver (Section 4.2.2) about symbol-address mappings through a VMCALL with the command REGISTER_STIR_SYMBOLS, and the address and length of an array of <symbol id, address> tuples. The return value of this VMCALL indicates success or failure.

Our implementation of the Symbol Mapper first performs a filtering of available kernel and module symbols before invoking the VMCALL, such that only STIR-related symbol-address mappings are passed to the Symbol Re-

solver. In order to perform the filtering, the STIR Symbol Mapper is initialized with a static list of STIR-related symbols, which is derived from the symbolic STIR summary signatures generated by the STIR Analyzer (Section 4.1).

4.2.2 Implementation of the STIR Symbol Resolver

As discussed in Section 3.2.4, the STIR Symbol Resolver is the security VM component to support STIR related symbol registration. The main task of this component is to handle the REGISTER_STIR_SYMBOLS command from the guest VM. It first copies the STIR-related symbol mappings (in a list of <symbol id, address>) from the guest kernel using the XenAccess [16] virtual machine introspection library. Next, it merges the addresses in the mappings to the STIR summary signature database (Figure 3) for that guest, using the symbol id as a search index.

In our implementation, each guest has its own instance of the STIR summary signature database. This database is initialized by a template generated from the STIR Analyzer (Section 4.1), where the addresses of the function symbols are undefined (the signatures are initially symbolic signatures). When the REGISTER_STIR_SYMBOLS command is executed, these symbols are resolved, and the corresponding signatures become resolved STIR summary signatures.

The symbol-address mapping registration must be carried out in a secure way, to ensure that the malware is unable to register a malicious callback function. Therefore we assume that some other measure is taken to ensure that this registration is performed only when the guest OS is in a "known good" state. Since a guest OS is less likely to be compromised in the early stage of its life (e.g., during a secure boot [2]), our current implementation approximates this requirement by dividing the life time of a guest OS into a *symbol registration phase* (i.e., the initialization time in Figure 3) followed by a *guarding phase* (the evaluation time in Figure 3), where symbol mappings can be registered only in the *symbol registration phase* (during this phase the guest OS is assumed to be in a "known good" state). We further perform the phase transition automatically for the guest kernel when it performs such registration for the first time, which is intended to minimize the attack window where a malware can misuse the VMCALL interface to insert malicious address mappings. However, a side effect of this implementation decision is that all legitimate LKMs that use soft timers must be loaded prior to the registration phase.

We note that this requirement may be undesirable for on-demand kernel module loading, but it can be resolved by other implementation options, such as verifying the runtime integrity of the guest kernel using Copilot [18] before allowing symbol mappings to be registered for a second time. Addressing these issues would improve the usability of the system, but security is already assured based on our assumptions. For these reasons, the usability issues are beyond the scope of this paper.

```
boolean check_stir (unsigned long function, unsigned long data){
    Use function as index to look up the resolved STIR summary signature database.
    If no signature is located, return false.
    Otherwise, if the assertion part of the located signature is empty, return true.
        Otherwise, return assertion (data).
}
boolean assertion (unsigned long data){
    for each predicate (deref equals functionlist){
      if deref(data) matches no address in functionlist
          return false.
    }
    return true.
}
```

Figure 6: Pseudocode of check_stir.

4.2.3 Implementation of the STIR Checking

As shown in Figure 5, the current STIR Checker is implemented in a security VM running on Xen. Its core function is check_stir, which performs verification of pending STIRs. As Figure 6 shows, check_stir takes as input two integer parameters: function and data, and returns true (success) or false (failure). It uses the resolved STIR summary signatures that are transformed from symbolic STIR Signatures by the STIR Symbol Resolver (Section 4.2.2).

The function deref(data) in Figure 6 uses the APIs provided by XenAccess [16] to dereference the data pointer (data) passed from the guest kernel (e.g., data->tx_timeout). The offset information is statically computed by using the output of the STIR Analyzer. For example, in order to dereference data->tx_timeout, where data is of type struct net_device *, we statically compute the offset of the field tx_timeout by analyzing the definition of struct net_device.

Finally, the soft timer dispatching logic of the guest Linux kernel is modified to make a VMCALL into Xen. Specifically, when a STIR in the pending timers queue expires, the guest kernel invokes a VMCALL, with the command CHECK_STIR, plus the function and data fields of the STIR as parameters. If the VMCALL returns true, function is called as normal. Otherwise, a warning message is generated and function is not invoked.

4.3. Evaluation of Linux Case Study

4.3.1 Effectiveness of Malicious STIR Detection

To evaluate the efficacy of our approach, we experimentally confirmed that our implementation of the STIR Checker is able to detect the key logger, the CPU cycle stealer and the alter-scheduler discussed in Section 2. We first installed our three "malware" kernel modules into an unprotected guest Linux kernel and confirmed that they are able to achieve their intended malicious purposes (e.g., stealing key strokes). We then activated the STIR-Aware environment containing the modified guest kernel, the Lares-patched Xen VMM, and the security VM running the STIR-Checker. We first instructed the STIR Symbol Mapper in

the guest kernel to register symbols with the STIR Symbol Resolver; currently this is initiated by loading the Symbol Mapper LKM. Then we installed the malware kernel modules. The STIR Checker is able to immediately generate warnings about the suspicious STIRs used by the newly loaded modules, and the malware functions are not invoked by the guest kernel as a result. The "malware" modules have been implemented using both attack techniques mentioned in Section 3. These results confirm that our approach can stop both types of STIR attacks.

False Positives. Under the assumption that the STIR Analyzer processes the complete source code of the guest kernel (including all legitimate modules), and the guest kernel installs all necessary and legitimate modules before registering symbol-address mappings, our detection has no false positives. This is because all potential legitimate STIRs have been captured in the resolved STIR summary signature database before the guest Linux enters the *guarding phase* (Section 4.2.2).

False Negatives. Due to our detection methodology, in order to obtain control, the malware must reuse legitimate STIR callback functions (such as dev_watchdog in Figure 2), and manipulate the parameter passed to the STIR callback function in such a way that control will eventually go to its malicious code. One way to leverage dev_watchdog has been shown in Figure 2. However, our detection techniques counter this type of attack by calculating and verifying the legitimate functions that can be assigned to dev->tx_timeout as shown in Table 3.

However, it is possible for the malware to search deeper in the control flow for opportunities, such as looking at the function ace_watchdog in Table 3, since ace_watchdog takes dev as the input parameter. This approach will also fail because the transitive closure analysis covers this case.

In summary, we believe that our detection can have no false negatives under the threat model in Section 3.1. However, since we may be facing a powerful adversary, our detection is not a panacea. A determined attacker may find a way not covered by our threat model to evade detection, although the STIR checking clearly raises the bar for an attacker.

Attacks on the STIR checking mechanism and our counter-measures. We anticipate that attackers may use either of two different kinds of attacks in an attempt to defeat the STIR checking. (1) The malware may disable the modification to the soft timer dispatcher so that it does not make the VMCALL, or ignores the return value. We protect against this by using Lares to make the code page of the soft timer dispatcher read-only. (2) The malware may try to register false mappings for legitimate symbols. By performing the phase transition (Section 4.2.2), such actions are ignored and therefore have no effect.

4.3.2 Performance Overhead

In order to measure the performance overhead of the STIR Checker, we ran a set of synthetic workloads: *cat* - read and display the content of 8000 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[1] 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. *make* - perform a full build of the Apache HTTP server (version 2.2.2) from source.

Table 4 shows the execution times of the workloads under Linux and our modified Linux (denoted STIR-aware). The VMM used in these experiments is Xen 3.0.4, and the guest Linux kernel is version 2.6.16. The host CPU is an Intel Core 2 Duo running at 2.4 GHz with VT-x enabled. The host and the HVM (i.e., fully virtualized) guest are allocated 1.5 GB and 512 MB of RAM, respectively.

	cat	ccrypt	gzip	cp	make
Original	20.85	3.30	5.92	43.95	217.95
STIR-aware	20.96	3.30	6.01	46.61	218.58
Overhead	0.52%	0%	1.52%	6.05%	0.29%
Callbacks/Sec	46.9	46.3	47.3	61.4	81.6

Table 4: Overhead measurement of the STIR Checker in execution time (seconds)

From Table 4 we can see that the performance overhead of the STIR Checker on the synthetic work-loads is low (less than 7%). Our testing found that out of the 365 STIR callback functions identified by the STIR Analyzer, only 74 are present in the guest kernel at runtime, and the majority of these STIR callbacks are dormant most of the time (although there may be multiple STIRs sharing the same call back function), therefore the frequency that a STIR actually expires (e.g., the frequency of the callbacks) is not high. For example, the baseline frequency of callbacks is around 45 per second. Table 4 shows the average frequency of callbacks during the experiment, which is similar to the baseline frequency.

We also evaluate the overhead of the STIR Checker by running the Iperf-2.0.2 benchmark[2]. In this experiment the security VM ran the Iperf server, and the guest VM ran the Iperf client. Iperf is used to measure the maximum throughput between the virtual NIC in the guest VM (the front end) and the virtual bridge in the security VM (the backend). The experiment is run for 60 seconds, using 64KB buffers and 10 concurrent connections. The average throughput in the original environment is 717.9 MB/s, and it is 688.4 MB/s in the STIR-Aware environment. This suggests a performance drop of 4.1% (decrease in network throughput). In addition, we measured the frequency of STIR callbacks during the Iperf experiment and found that it increased to 287 per

[1] http://sourceforge.net/projects/ccrypt/
[2] http://dast.nlanr.net/Projects/Iperf/

second, which explains the slightly higher overhead of the STIR Checker compared to the synthetic workloads.

In summary, the performance overhead for the STIR-Aware environment is small compared to the added security benefit that it provides.

5. Related Work

Defenses Against Stealthy Attacks. Defense techniques against attacks that change kernel code include Tripwire [13], a file system integrity checker, IMA [22], a load-time kernel and application code integrity checker, and Copilot [18] and Pioneer [25], runtime kernel code checkers. Representative defenses for attacks that change kernel data include CFI [1] and SBCFI [20].

To the best of our knowledge, there have been few concrete instances of attacks that do not change kernel code or data, but insert transient execution units into a schedulable queue. The "cheat" attack described in [27] may be regarded as a user-level example, since it uses the to-be-scheduled task queue. Known malware detection methods have difficulties with transient kernel control flow attacks. For example, signatures of known malicious STIRs can be created by reverse engineering the malware. This approach suffers from the same problems seen in the anti-virus community. Specifically, they are unable to detect or prevent zero-day attacks, and the process of finding appropriate signatures is difficult and error prone. For these reasons, signature checking alone is insufficient to mitigate this threat.

Another possible approach for detecting these attacks is to extend control flow integrity techniques such as SBCFI [20] and CFI [1]. SBCFI is a checker for persistent kernel control flow attacks. It starts by looking at kernel global variables and performs 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 STIR, since the function pointer targets can be validated when SBCFI scans the kernel variables. However, SBCFI can not detect type 2 STIRs because it does not follow the uninterpreted *data* field included as part of the callback: it is not defined as a pointer type. The definition of data is intended to allow maximum flexibility for different call back functions. In order to make SBCFI work on type 2 STIRs, accurate type information for the data field in each call back function must be added, which would require a static analysis of all STIR callback functions. Such an approach would then be similar to our STIR Analyzer (Section 3.2).

A more general approach, CFI [1] uses inline reference monitors [11] to compare the dynamic execution flow of a program against a statically computed control flow graph (CFG). CFI is a general framework that can be instantiated into an alternative implementation of the STIR Checker, however the exact checks that must be performed against the STIR callback functions would still need to be constructed by tools such as the STIR Analyzer.

Secure Kernel Extensions. Soft timer-driven malware exploits an interface exposed by the core kernel to its extensions. There have been significant efforts to achieve finer-grained divisions within a monolithic kernel, with the goal of improving security. For example, Palladium [6] demotes the privileges of the kernel extensions so that misbehaving or malicious extensions cannot harm the core portion of the kernel. However, such approaches can only prevent the malicious extensions from corrupting the core kernel, but cannot prevent sensitive information stealing (section 2.1) and denial of service attacks (Section 2.2).

Applications of Static Analysis in Systems and Security Research. In recent years, static analysis of software has been used for many purposes including deriving application behavior models for intrusion detection systems [28], building control flow graphs of an application [1], and determining type and global variable information for the Linux kernel [20]. This technique has also been applied to finding bugs in both kernel and application code [7, 10, 14]. In this paper, we add one more use case by applying this technique to derive summary signatures for legitimate STIRs.

6. Conclusion

In this paper, we introduce the class of *transient* kernel control flow attacks that control a kernel through soft timer interrupt requests (STIRs), without changing kernel code or data (as in persistent kernel control flow attacks [20]). We demonstrated the attack by implementing prototype malware in Linux (Section 2), including a key logger, a CPU cycle stealer, and an alter-scheduler, that demonstrate effective exploitation of soft timer requests to violate integrity, confidentiality, and availability of the kernel. Mechanisms effective in detecting persistent kernel control flow attacks such as SBCFI [20] have difficulties with STIR-based attacks, since these transient kernel control flow attacks preserve kernel code and data integrity.

Due to the widespread use of soft timers in typical kernels such as Linux, it is impractical to eliminate soft timers just to stop these attacks. Consequently, we designed a VMM-based defense mechanism against STIR-based transient kernel control flow attacks. The main idea of the defense is to analyze the entire kernel and build a summary signature database of legitimate STIRs. In Linux 2.6.16, a total of 365 legitimate STIR callback functions were found in the 3688 kernel source files analyzed. 32 of the 365 STIRs may further branch to function pointers derived from their input parameters, requiring a more detailed transitive closure analysis of the functions invoked by these 32 STIRs. If a STIR can be found in the signature database, then it is considered legitimate and executed. Otherwise, it is considered suspect and rejected. This defense is effective in stopping each of the prototype malware STIRs we implemented (no false negatives), and it allows all legitimate STIRs to execute (no false positives). The performance overhead of our defense is small (less than 7%) in several application-level

benchmark experiments.

Acknowledgement

This research has been partially funded by National Science Foundation grants ENG/EEC-0335622, CISE/CNS-0646430, CISE/CNS-0716484, CNS-0627430, AFOSR grant FA9550-06-1-0201, NIH grant U54 RR 024380-01, IBM SUR grants and a Faculty Partnership Award, Hewlett-Packard, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[4] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly, Dec. 2002.

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

[6] T. c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Charleston, SC, Dec. 1999.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Nov. 2002.

[8] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Charleston, SC, Jan. 1993.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, TX, Jan. 1998.

[10] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[11] U. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[12] S. Hultquist. Are rootkits the next big threat to enterprises? `http://www2.csoonline.com/blog\_view.html?CID=32882`, Apr. 2007.

[13] G. Kim and E. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *2nd ACM Conference on Computer and Communications Security (CCS)*, Fairfax, VA, Nov. 1994.

[14] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

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

[16] B. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2007.

[17] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

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

[19] N. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, Vancouver, BC, Aug. 2006.

[20] N. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2007.

[21] rd. Writing linux kernel keylogger. *Phrack*, 59(11), June 2002.

[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[23] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), Sept. 1975.

[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[25] A. Seshadri, M. Luk, E. Shi, a. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, Oct. 2005.

[26] Solar Designer. Getting around non-executable stack (and fix). *Bugtraq*, Aug. 1997.

[27] D. Tsafrir, Y. Etsion, and D. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *USENIX Security Symposium*, Boston, MA, Aug. 2007.

[28] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.