Automated Detection of Persistent Kernel Control-Flow Attacks

Department of Computer Science Technical Report CS-TR-4880

Nick L. Petroni, Jr. and Michael Hicks
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742 USA
{npetroni,mwh}@cs.umd.edu

ABSTRACT

This paper presents a new approach to dynamically monitoring operating system kernel integrity, based on a property called *state-based control-flow integrity* (SBCFI). Violations of SBCFI signal a persistent, unexpected modification of the kernel's control-flow graph. We performed a thorough analysis of 25 Linux rootkits and found that 24 (96%) employ persistent control-flow modifications; an informal study of Windows rootkits yielded similar results. We have implemented SBCFI enforcement as part of the Xen and VMware virtual machine monitors. Our implementation detected all the control-flow modifying rootkits we could install, while imposing negligible overhead for both a typical web server workload and CPU-intensive workloads when operating at 1 second intervals on a multi-core machine.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

CFI, integrity, virtualization, rootkit, kernel

1. INTRODUCTION

Attackers that gain entry into computer systems frequently wish to inject illicit functionality to maintain control, gather information (e.g., keys pressed or packets sent), or neutralize the defenses of the target system, among other objectives [3]. While past attacks often focused on modifications to user-level libraries or system binaries, the operating system kernel is an increasingly popular target [16]. Attackers typically modify the kernel—using so-called "rootkits"—either to implement their illicit functionality directly, to hide its implementation in user-level objects, or both.

One way to detect kernel modifications is to verify expected invariants of the kernel's state; violation of an invariant suggests the kernel has been compromised. Invariants may be checked directly, e.g., by using a virtual machine monitor or a separate card to examine kernel memory [14, 37, 23], or may be checked indirectly via user-level tests [12, 8]. To date, most invariants have been designed to address known attacks [24]. For example, attackers have been known to modify the kernel's system call table to point to malicious code. To detect this attack, some systems regularly compare the contents of the system call table to a previously observed state [14, 23]; any difference suggests an attack. Unfortunately, once one attack target is closed off, adversaries simply find other targets, e.g., device driver jump tables. To end this cycle, we must devise properties that consider an attacker's higher-level goals, so that by checking those properties we can detect a wide range of attacks.

Based on an extensive analysis of existing rootkits, we believe that one property that meets this challenge is *control-flow integrity* (CFI) [1]. This property dictates that software execution must follow paths according to a control-flow graph (CFG) determined in advance; system call table modifications are one instance of a CFI violation. More generally, the goals of rootkits are squarely at odds with CFI. Since the attacker's main goal is to add surreptitious functionality to the system, then either this functionality or the means to hide it are most easily enabled by modifying the kernel's CFG to call injected code. Of 25 Linux rootkits we studied, we found that 24 (96%) violate CFI.

This paper presents a technique for enforcing an approximation of CFI that we call *state-based CFI* (SBCFI). CFI, as originally proposed, is implemented using an in-line reference monitor [11] in which checks are added to the target program to validate each dynamically-computed branch during execution. We observe that rootkit writers are motivated to make *persistent* modifications to the CFG, e.g., to add illicit functionality that performs keystroke logging or packet analysis, or to hide such functionality when it is implemented in user-level objects. Therefore, rather than validating every branch as it happens, we periodically examine the kernel's state and validate it as a whole. This approach has two useful properties: (1) we can implement the validation in an external monitor, which is more easily

[0]This version supplements the version appearing at ACM CCS 2007 by including a more complete performance analysis. As such, Section 4 reflects an updated set of performance experiments including new hardware, complete SPECCPU2006 INT benchmarks, and numbers for both VMware and Xen virtual machine environments. Other references to that section have also been updated accordingly.

protected from tampering; (2) we can trade off precision for performance by altering monitoring frequency. Since most CFG-modifying attacks are persistent, monitoring can be infrequent enough to have essentially no overhead.

We have implemented SBCFI as part of the Xen and VMware Workstation virtual machine monitors. The state analysis proceeds in two steps. First, we check that none of the kernel code has been modified, validating all static branches. Second, we perform a garbage collection-style traversal of the heap to find all usable function pointers, and verify that they target valid code. We generate the traversal code via a simple analysis of the kernel source. Our implementation supports dynamically-loadable kernel modules.

We applied our implementation to a default installation of the RedHat Linux 7.3 distribution. Of the 25 Linux rootkits we studied, we could install 18 on this platform and our SBCFI monitor detected all of those that modified the kernel's control-flow in some way (17 out of 18). Furthermore, our SBCFI monitor added negligible overhead on top of that imposed by the VMM when monitoring every second on a multi-core system. The VMM overheads were themselves often negligible, with a worst-case overhead of around 9%. We expect VMM overheads to continue to decrease given their popularity, particularly in data centers [4].

In summary, this paper makes the following contributions:

- We show that enforcing control-flow integrity can potentially defeat a large class of kernel integrity violations. We studied 25 known Linux rootkits in detail and found that 24 (96%) violate CFI. We further argue that future rootkits have incentive to violate CFI, based on typical attacker goals. (Section 2)
- We describe a technique we call *state-based control flow integrity* (SBCFI), which implements an approximation of CFI via periodic analysis of the kernel's state (Section 2.2). The monitor can be kept separate from the target kernel, to protect it from tampering, and it can detect rootkits that make persistent changes to the CFG. All 24 of the CFI-violating rootkits we studied also violate SBCFI.
- We describe our implementation of state-based CFI using the Xen virtual machine monitor (Section 3), and we evaluate its effectiveness and performance against Linux-based rootkits (Section 4). We show that we can defeat all control-flow modifying rootkits in our corpus, and that monitoring at intervals of 1 second or more on a dual-core system imposes negligible overhead on top of the cost of the VMM (which itself imposes at most 9% overhead on our benchmarks, but typically much less). To our knowledge, our SBCFI monitor is the most effective kernel integrity monitor proposed to date. Prior monitors limit the scope of their checks to easy-to-find function pointers or invariants targeted by specific known attacks; they may be difficult to apply to real systems; or they impose unacceptably high overhead. (Section 5)

We begin by describing threats to kernel integrity and proceed to motivate state-based CFI as a means to detect a wide array of integrity-violating attacks.

2. MONITORING KERNEL INTEGRITY

In an attempt to introduce an illicit capability into a computer system, an attacker may wish to modify the operating system kernel K. The goal of the modification may be to directly implement (part of) the illicit capability, to hide its implementation in one or more user-level objects, or both. We call such a modification an *integrity violation*. An attack against the kernel's integrity is called a *rootkit*.

Table 1 contains a list of Linux kernel rootkits that we were able to obtain and analyze. The left section of the table presents the set of attack goals that we identified for each rootkit. Based on our analysis, the objectives of each rootkit fall into one or more of the following categories: user-space object hiding (HID), privilege escalation (PE), reentry/backdoor (REE), reconnaissance (REC), and defense neutralization (NEU). Hiding and privilege escalation facilitate a user-space implementation of an illicit capability, while the latter three implement some or part of an illicit capability directly in the kernel.

The goal of an *integrity monitor* M is to detect integrity violations of K. In the ideal case, M would detect violations by verifying that K is behaving properly; any improper behavior would suggest an integrity violation. Because checking complete behavioral correctness is extremely difficult, a more practical goal is for the monitor to check a specific property p such that, if K is correct, then p holds. Conversely, if p fails to hold, then K must be behaving improperly. The challenge is choosing a p that considers the attacker's goals and addresses available mechanisms for achieving those goals in the kernel. In other words, we want to choose p such that attacks are likely to violate it. We believe one such property is *control-flow integrity* [1].

2.1 Control-Flow Integrity

A program P satisfies the CFI property so long as its execution only follows paths according to a *control-flow graph* (CFG), determined in advance. If this graph approximates the control flow of the unmodified P, then a violation of CFI signals that P's integrity has been violated. CFI enforcement has been shown to be effective against a wide range of common attacks on user programs, including stack-based buffer-overflow attacks, heap-based "jump-to-libc" attacks, and others.

Our analysis of Linux rootkits shows that an overwhelming majority of them, 24 out of 25 (96%), violate the control-flow integrity of the kernel in some way. As far as we are aware, we are the first to make this observation. Additionally, our preliminary analysis of about a dozen Windows kernel rootkits demonstrates a similar trend among those threats. This suggests that CFI is a useful property to monitor in the kernel.

In Table 1, the middle group of columns characterizes the changes made by each rootkit. For those attacks that make some modification to the kernel's control-flow, a mark was placed in at least one of the columns under the heading *Control-flow Mods*. Those threats that make changes to non-control data [7] are marked in the adjacent column. (The meaning of the marks P, B, and T is explained shortly.) We categorize control-flow modifications further according to which type of object is modified during the attack: kernel text (labeled *Text*), processor registers (labeled *Reg*), or function pointers (labeled *FP*). All rootkits but `hp` violate CFI.

Attack name	Functionality					Control-flow mods			Non-control data mods	Testing
	HID	PE	REE	REC	NEU	Text	Reg.	FP		
Adore	X	X						P	P	D
Adore-ng	X	X			X			P	P	D
All-root		X						P		-
Anti Anti Sniffer	X			X				P		-
enyelkm	X	X	X			P				-
hp	X	X							P	N
kdb	X	X						P		D
KIS	X		X					P	P	D
Knark	X	X	X					P	P	D
Linspy			X					P		D
Maxty				X				P		-
Modhide	X							P	P	D
Mood-nt	X			X		B	P	B		D
Override	X	X						P		-
Phantasmagoria	X					P			P	D
Phide	X							P		-
Rial	X							P		D
Rkit		X						P	P	D
Taskigt		X						P		-
Shtroj2		X						P		D
SucKIT	X			X		P		T		D
SucKIT2	X	X	X	X				B		D
Superkit	X		X	X		P		T		D
Synapsys	X							P		D
THC Backdoor		X						P		D

HID: hiding PE: privilege escalation REE: reentry REC: reconnaissance NEU: neutralization
X: provides functionality P: persistent T: transient B: both P and T
D: detected in testing N: did not detect - (dash): unable to test

Table 1: Analyzed kernel threats: functionality and mechanisms.

We wish to enforce CFI under the assumption of a powerful adversary who has arbitrary read and write access to kernel memory. Such an assumption is not unreasonable. Loadable kernel modules (LKMs), which are common in both Windows and Linux, have complete access to the kernel's address space, and may be loaded in response to events not directly under a user's control. Buffer overrun and other vulnerabilities afford the attacker the ability to corrupt possibly arbitrary areas of kernel memory. Compromised hardware devices, and even standard virtual devices, such as Linux's `/dev/kmem` or portions of its `/proc` file system, may also provide the attacker access to kernel memory.

Abadi et al. [1] enforce CFI for a program P by rewriting P's binary. The target of each non-static branch is given a tag, and each branch instruction is prepended with a check that the target's tag is in accord with the CFG. This strategy provides protection against an attacker with access to P's memory under three conditions: (1) tags must not occur anywhere else in P's code; (2) P's code must be read-only; and (3) P's data must be non-executable. These assumptions are easily discharged for applications. The first assumption is discharged by rewriting the entire application at once, preventing conflicts, and the latter two are discharged by setting page table entries and segmentation descriptors appropriately; as the page tables can only be modified within the kernel, it is assumed they are inaccessible to the attacker.

Unfortunately, these assumptions cannot be so easily discharged when monitoring the kernel itself. An attacker with access to kernel memory could overwrite page table entries to make code writable or data executable, violating assumptions (2) and (3). It is also unrealistic to expect to rewrite all core kernel code and LKMs at the outset, and thus it is difficult to discharge the first assumption and avoid tag conflicts. For that matter, it is nontrivial to compute a precise CFG for the kernel in advance, due to its rich control structure, with several levels of interrupt handling and concurrency.

2.2 State-based CFI Monitoring

To avoid these problems, we propose to enforce an approximation of CFI using a new technique called *state-based monitoring*; we call the resulting property *state-based control-flow integrity* (SBCFI).

A *state-based monitor* checks the system periodically, rather than in step with the program's execution. More precisely, a *state-based monitor* M_p^n signals a violation when, after K has taken n steps since it was last checked, it enters a state s such that p does not hold. This has two benefits. First, by analyzing K's states, and not its transitions, M_p^n is more easily separable from K, which makes M_p^n better protected from tampering. For example, M_p^n can be kept in a virtual machine or on a separate card. Second, M_p^n can be tuned to trade off performance and precision. Smaller

values of n have greater precision, while larger values of n have better performance. Despite not dealing with transitions directly, state-based monitoring can be effective because the program's subsequent execution possibilities are captured by its current state, i.e., its code and data. Analyzing this state, the monitor can determine whether p could be violated during later execution.

We enforce SBCFI by ensuring that the CFG induced by the current state is not different from the CFG of the initial kernel. The details of our implementation are presented in Section 3. In summary, our approach has two steps:

1. *Validate kernel text, including static control-flow transfers.* The monitor keeps a copy or hash of K's code. At each check, it makes sure K's code has not been modified by comparing it against the copy or hash. This ensures that static branches occurring within the kernel (e.g., direct function calls) adhere to the kernel's CFG.

2. *Validate dynamic control-flow transfers.* To validate dynamically-computed branches, the monitor must consider the dynamic state of the kernel—the heap, stack, and registers—to determine potential branch targets. Our implementation relegates its attention to function pointers within the kernel. Analogous to a garbage collector, the monitor traverses the heap starting at a set of roots—in our case, global variables—and then chases pointers to locate each function pointer that might be invoked in the future. It then verifies that these pointers target valid code, according to the CFG.

Because it monitors K's state periodically, an SBCFI monitor can only be used to reliably discover *persistent* changes to K's CFG: if an attacker modifies the kernel for a short period, but undoes his or her modifications in time less than n, then M_p^n may fail to discover the change.

Nevertheless, limiting our attention to persistent modifications to the CFG is extremely useful. Consider Table 1 again. For each modification that we identified in columns 7–10, we also determined whether the modification is persistent, represented by a P, or transient, represented by a T. Columns with a B indicate that both persistent and transient modifications of a given type are made. We found that all of the analyzed threats make some persistent change to the kernel. Furthermore, in all of the attacks in which a control-flow modification is made, some portion of those changes was found to be persistent. Thus the same 24 rootkits that violate CFI also violate SBCFI.

This makes sense from the attacker's perspective: the goal of a rootkit is to introduce surreptitious, long-term functionality into the target system, e.g., to facilitate later reentry, reconnaissance (keystroke monitoring, packet sniffing, etc.), or defense neutralization. Changes to the kernel CFG to facilitate this are thus *indefinite*, so SBCFI will discover them, even for large values of n.

As an example of how an attacker might persistently modify the control-flow of the kernel, consider the `Linspy` Linux rootkit. `Linspy` is a kernel-level keystroke logger that modifies the kernel in two ways. First, it redirects the `write()` system call to look for write events from processes of interest. Second, it creates a new character device, e.g., `/dev/linspy`, to provide a malicious userspace process with access to the collected data. The latter is performed by making a simple function call to the kernel's `register_chrdev()` function. `Linspy` results in five SBCFI violations – one for the modified system call, and four for the registered callback functions (`open()`, `release()`, `read()`, `ioctl()`) associated with the added character device.

The only attack that we encountered that does not cause a persistent control-flow modification is the `hp` rootkit, and therefore SBCFI will not detect it. `hp` performs simple process hiding by removing the process from the kernel's "all tasks" list (while leaving it in its "to-be-scheduled tasks" list). Other attacks utilize a similar technique for hiding modules, but must make control-flow modifications elsewhere to enable execution of the module's code.

While an SBCFI-based monitor detects many attacks that would not be detected by previous kernel integrity monitors, it is not a panacea. While current attacks always inject some persistent modification, an attacker could avoid detection by persistently applying transient attacks. For example, a remote host could regularly send a packet that overruns a buffer to inject some code which gathers local information and then removes itself, all within the detection interval. Even so, this form of attack limits what the attacker can do compared to having persistent code in the kernel itself, e.g., to log keystrokes. A clever attacker might find a way to corrupt a kernel data structure so that periodic processing in the kernel itself precipitates the buffer overrun. Though much harder to construct, such an attack would help avoid detection via a network intrusion detection system, and could make information gathering more reliable.

Our implementation of SBCFI is also limited because we relegate our attention to function pointers and not other forms of computed branch, such as return addresses on the stack, or in the extreme case, function pointers manufactured by some complex computation. Not considering return addresses prevents detection of stack smashing attacks, but since these attacks are typically transient this is less of an issue. We could miss modifications to portions of the stack that are long-lived. To our knowledge, function pointers in the kernel are usually stored in record fields directly, and not computed.

In short, though SBCFI may miss attacks that would be detected by CFI, it is straightforward to build a SBCFI monitor that is protected from tampering; that can detect the types of mechanisms used by all of the control-flow modifying rootkits we could find; and that significantly "raises the bar" for constructing new attacks.

3. IMPLEMENTATION

To evaluate the usefulness of our approach, we implemented a state-based CFI monitor for the Linux kernel. To protect it from tampering, the kernel monitor operates separately from the target kernel being monitored, using a virtual machine monitor (VMM)[14]. We have developed monitors based on both the Xen [5] and VMware [34] VMMs. Figure 1 illustrates the Xen implementation. In this configuration, the VMM runs on top of the bare metal and supports two virtual machines, one for the monitor and one for the target. A process in the monitor VM performs the SBCFI checking of the target, calling into the VMM to access to the target VM's memory and registers. The VMware implementation runs a single VM containing the target kernel, with the VMM itself running as a process within a host operating system. The SBCFI monitoring process likewise

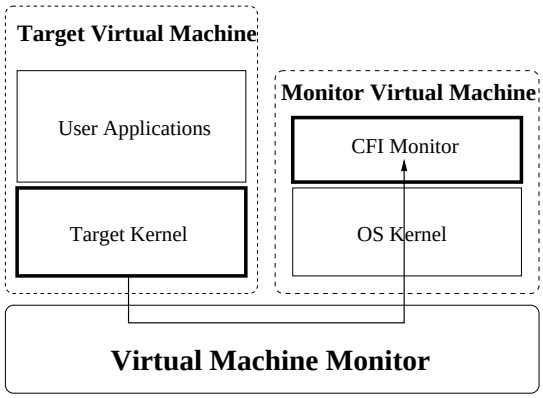

Figure 1: SBCFI runtime monitor setup.

runs within the host OS, calling into the VM to access the target's state.

Much of the monitor process is generated automatically from the target kernel's source code and compiled binary, as shown in Figure 2. The generation proceeds in several stages. The *Type & Global Extractor*, *Symbol Manager*, and *Type Mapper* are used to gather information about the target kernel's symbols and type structure. This information is passed as input to the *Monitor Generator*, a Python program that generates C code to traverse the kernel's heap to look for function pointers. The generated code is linked against VMM-specific routines in the monitor library for accessing the target's memory.

As described in Section 2.2, to verify that the kernel's control-flow has not been modified, the kernel monitor performs two tasks: (1) it validates that the kernel's text has not been modified and (2) it verifies that all reachable function pointers are in accord with the kernel's CFG. We discuss each of these points in turn.

3.1 Validating Kernel Text

For the Linux kernel, the set of allowable runtime code is determined by two sources: (1) the static portion of the kernel that is loaded by the boot loader and (2) a set of authorized loadable kernel modules (LKMs), which can be loaded or unloaded dynamically during kernel execution. The generated monitor takes as input trusted copies of the kernel and LKM binaries for runtime comparison (this is shown by the dashed line in Figure 2). The code verification procedure works as follows:

1. Compare the executable sections of the static kernel in the trusted store with those the in memory. If equal, add the sections to the set of verified code regions V; otherwise, add them to the invalid code regions set I.

2. Traverse the list of kernel LKMs kept by the target kernel[1] to locate all currently loaded modules. For each kernel module:

   (a) Locate the trusted copy of the LKM. If no trusted copy can be located, add the module to I.

   (b) Emulate the module loader to adjust all relocatable symbols in the trusted LKM copy based on where the module is loaded in target memory.

   (c) Compare the text sections of the emulated copy to what is in memory at the expected location. If equal, add the sections to V; otherwise add them to I.

3. Report the set of invalid code regions I if nonempty. (The set V is used in the next phase.)

Step 2b is necessary because LKMs are relocatable. When emulating dynamic linking, a module's external references to kernel symbols are resolved to what the monitor believes is potentially valid code and data; i.e., the targets must reside in the text or static data area of the core kernel or one of the modules in the module list. This avoids having to trust the kernel's module symbol table and has the effect of validating any static, inter-module function calls.

To make text verification more efficient, we applied two optimizations to this algorithm. First, we use a cryptographically secure hash algorithm to speed up all comparisons (Steps 1 and 2c). Second, we cache the hashes of the relocated, trusted LKMs computed in Step 2b. We can reuse these when comparing to in-kernel modules whose position has not changed since the last check.

3.2 Validating Dynamic Control-flow

Validating the kernel's text ensures that all static control-flow transfers—in particular, direct function calls—are in accord with the kernel's CFG. The monitor must also validate all dynamic control-flow transfers; i.e., those for which the transfer target is not known until run-time. For the x86 architecture, the two main sources of dynamic transfers are (1) indirect calls to functions (i.e., via function pointers) or labels (e.g., as part of a `switch` statement), and (2) function call returns (regardless of whether the function was called directly or indirectly). The latter category is typically implemented by popping a return address off of the stack and jumping to it, e.g., via the `return` instruction.

As already discussed, our kernel monitor does not consider function call return targets or intraprocedural dynamic branch targets. This is because such attacks, in and of themselves, do not create a persistent integrity violation in the kernel. This leaves us with the task of verifying the targets of function pointers that might be used by the kernel during later execution. The first step is to identify the set of possible function pointers. A reasonable approximation of this set is those function pointers *reachable* from a set of roots via a chain of pointer dereferences, in the spirit of a garbage collector (GC). We can construct a traversal algorithm to find the reachable function pointers, given three inputs: (1) the set of initial roots (e.g., global variable addresses, the stack, and the registers); (2) the offsets within each object at which there are pointers; and (3) an indication of which pointers within an object are function pointers. With this, a traversal algorithm can start at the roots and transitively follow the pointers embedded in objects it reaches until all function pointers have been discovered.

We gather the necessary inputs via static analysis of the kernel's source code and compiled binary, and the monitor generator constructs the traversal code in three steps:

1. From the kernel source, extract all global variables and their types (Section 3.2.1).

[1]The address of the root of this list is determined by examining the trusted static kernel binary.

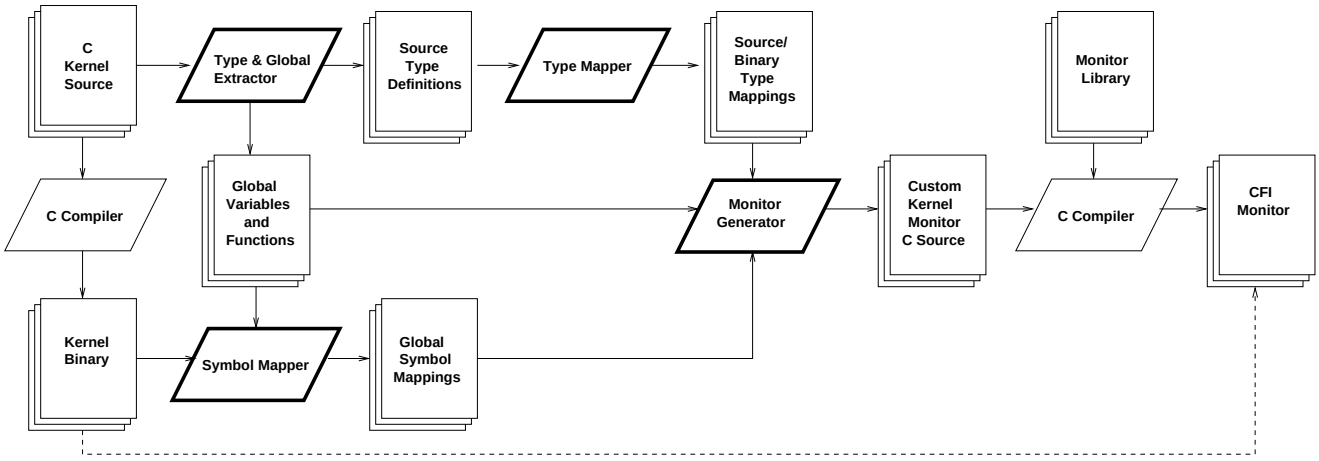

Figure 2: Monitor generation process and components.

2. Construct a *type graph* based on the type definitions occurring in the kernel source. Each node in the graph represents a type, and an edge from T_1 to T_2 implies objects of type T_1 contain a pointer (or pointers) to objects of type T_2. The graph includes only types from which function pointers can ultimately be reached (Section 3.2.2).

3. Using the global variables as starting points and the type graph as a specification, generate code to locate all function pointers reachable from global variables (Section 3.2.3).

As it discovers reachable function pointers, the traversal algorithm will then validate those pointers according to an approximation of the CFG, described in Section 3.2.4.

3.2.1 Finding the Roots

The first step in generating the traversal code is to identify the roots, which include the program's global variables, the stack, and the registers. The *Type & Global Extractor* (Figure 2) extracts the global variables and their types from the kernel source using a simple C Intermediate Language (CIL) [21] module.

We do not consider the stack and most of the registers in our traversal code because, unlike global variables, their contents cannot be given a static type: they will contain values of different types depending on the current program counter and calling context. To address this issue in a garbage collection setting, the compiler can generate metadata used by the GC traversal to designate the types of the registers and stack frames at various program points. Constructing such a compiler for C would be a substantial undertaking, and would be complicated by C's weak type system (discussed below). In the absence of such data, a conservative garbage collector [6] can pessimistically regard any stack or register word as a pointer if it falls within the range of legal memory (among other validation criteria). In our setting this approach is insufficient as we also need to know the *type* of that memory, to know whether it contains function pointers that we must validate. We consider the x86 registers `idtr`, `gdtr`, `sysenter`, the debug registers, `eip`, and the `cr` registers as roots because the type of their contents is fixed.

procedure BUILDTYPEGRAPH()
$Nodes \leftarrow$ set of extracted types from kernel source
$Edges \leftarrow \emptyset$
$FPNodes \leftarrow \emptyset$
for each Struct $s \in Nodes$
  do **for each** member $m \in Members(s)$
    do **if** $IsFunctionPointer(m)$
      then $FPNodes \leftarrow FPNodes \bigcup \{s\}$
    if $IsStruct(m)$
      then $Edges \leftarrow Edges \bigcup (s, Type(m))$
for each Struct $n \in Nodes$
  do $ExcludeNode \leftarrow$ **true**
    for each Struct $f \in FPNodes$
      do **if** $PathExists(Edges, n, f)$
        then $ExcludeNode \leftarrow$ **false**
          break
    if $ExcludeNode$
      then $RemoveNode(Nodes, Edges, n)$
return $(Nodes, Edges)$

Figure 3: Algorithm to generate type graph

Not considering the complete root set means we will miss some CFG modifications; e.g., we will not notice modified code pointers on the stack or in untyped registers, nor will we notice modified code pointers in objects reachable only from these locations. Nevertheless, because the contents of the stack and the untyped registers are transient, ignoring them should not cause us to miss persistent attacker modifications.

3.2.2 Constructing the Type Graph

The next step is to construct the type graph. This happens in two steps. First, the *Type & Global Extractor* extracts all type definitions from the kernel source. With these as input, the *Monitor Generator* builds the type graph $G(n, e)$ using the procedure in Figure 3. Figure 5 depicts the type graph for the (simplified) types found in Figure 4, with the function-pointer containing structures highlighted (`den-`

```
// @head, @type super_block(s_list)
struct list_head g_super_blocks;

struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
struct fs_struct {
    struct dentry   *root;
    struct vfsmount *rootmnt;
};
struct dentry {
    struct dentry            *d_parent;
    struct inode             *d_inode;
    struct dentry_operations *d_op;
}
struct inode {
    struct inode_operations *i_op;
    struct file_operations  *i_fop;
    struct super_block      *i_sb;
};
struct super_block {
    // @headed, @type super_block(s_list)
    struct list_head         s_list;
    struct dentry           *s_root;
    struct super_operations *sop;
};
struct vfsmount  {
    struct vfsmount    *mnt_parent;
    struct dentry      *mnt_root;
    struct super_block *mnt_sb;
};
```

Figure 4: Simplified Linux type definitions.

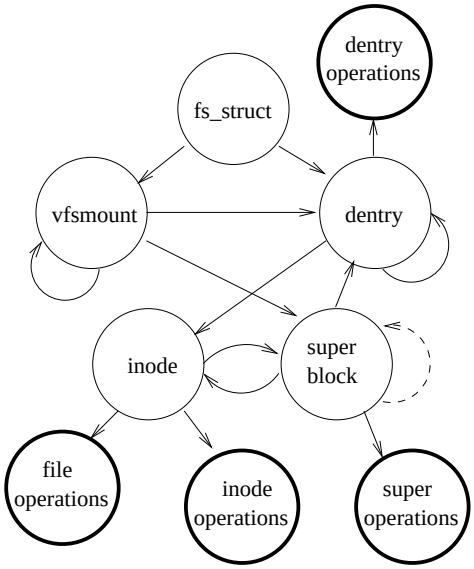

Figure 5: Function pointer reachability graph.

try_operations, inode_operations, super_operations, and file_operations; definitions not shown in Figure 4).

Unfortunately, the type information in the kernel source is not sufficient to identify all typed pointers. Because C's type system is not expressive enough to specify some useful idioms, programmers use conventions to encode them. For example, C does not provide polymorphism (generics), so programmers often cast generic elements to/from `void*` (or even `int`). Similarly, the Linux kernel makes heavy use of list data structures embedded in other objects, and the precise type of the target object of each `next` pointer is not evident from the static type. There is also insufficient static type information to disambiguate the current value of an untagged `union` or the size of a dynamically-sized array.

We can overcome these limitations with user-provided annotations. For the purposes of our experiments, we have annotated the embedded list cases (described below), but left the arrays, unions, and other cases to future work; these would probably have annotations similar to those provided by Deputy [9, 38], with the advantage that they would only be required on type definitions and not function declarations. Because we do not annotate arrays, unions, and manufactured or generic (`void*`) pointers, we may not find all reachable function pointers and thus potentially miss some violations. Nevertheless, even with this limitation we are able to detect all control-modifying rootkits we could install on our test platform.

We describe our embedded list annotations by example. Consider the simplified `super_block` structure shown in Figure 4. Its member `s_list` is of type `struct list_head`, which contains fields that, according to the definition, link to other `list_head` objects. By convention, these other `list_head` objects are actually the `s_list` fields of other `super_block`s, allowing `super_block`s to be chained together into a linked list. Here, the linked list is headed by the global variable `g_super_blocks`, and the last element of the list will point to `g_super_blocks` itself, terminating the list. Traversal code may cast each `next` or `prev` pointer to a `super_block` to access its fields. Of course, such code must check, before performing the cast, that a pointer is to another `super_block` object by ensuring it does not point to `g_super_blocks`, the head/terminator of the list.

So that the monitor can properly traverse embedded lists of objects, we specify this convention using some simple annotations. We annotate each occurrence of a global variable or structure field of type `list_head` with the type of the objects into which the `list_head` actually points. If the pointer is into the middle of a structure, we also include the field name of the precise position; the start of the object can thus be recovered by subtracting `offsetof(`*`fname`*`)` from the `next` or `prev` pointer. In Figure 4, we have added comments including annotation *@type super_block(s_list)* above both the `s_list` member and the `g_super_blocks` global.

Our example illustrates a "headed" list, in which each embedded `list_head` could point to another `list_head` embedded within the given `@type`, or to the head/terminator of the list. We indicate this by annotating a list's head with `@head` (in the example, the `g_super_blocks` global variable is so annotated) while the `list_head`s within a headed list are annotated with `@headed` (in the example, the `s_list` fields are so annotated). To cast such a field to its given `@type` thus requires a check that the field does not point to the head. Alternately, to represent a non-headed list, we can annotate `list_head` occurrences as `@nohead`. This means that they will *always* point to objects of the given `@type`.

For our Linux 2.4.18-3 kernel, we annotated 123 type definitions and 39 variable definitions. The process was fairly straightforward and took us just under two days, working

sporadically, to complete. We believe that most of these annotations could be inferred—and many of the generic, array and union annotations as well—using a constraint-based analysis along the lines of CCured [22].

3.2.3 Implementing the Traversal

The final step is to use the type graph to generate the traversal code for the monitor. The generated code performs a modified breadth first search (BFS), starting at each global variable whose type appears in the type graph. When an object is visited, all function pointers (if any) that are part of the object are checked, and all neighbors (as determined by the type graph) are added to the queue of nodes remaining to be visited (nodes are marked so they are not revisited). The only exception to BFS ordering occurs when a node is reached that contains one or more list heads annotated with `@head`. In these cases, each list is traversed to completion, following the appropriate `@headed` link field in its members; `@headed` links are ignored except during such traversals. This approach ensures that all members are reached and treated in a type-correct manner. Non-headed list pointers (annotated `@nohead`) are treated like any other neighbor pointer in the graph, processed according to BFS.

Because the traversal will be run within a process outside of the target kernel, it requires a mechanism to map source-level type and variable definitions to their low-level representation in the running kernel. Specifically, two source→binary mappings are required. First, the monitor must know the virtual addresses of the running kernel's global variables. The *Symbol Mapper* (see Figure 2) extracts these from the kernel's binary files. Second, the monitor must be able to map source-level types to their binary representations in memory. The *Type Mapper* (implemented in C) uses the kernel compiler to generate this information from the kernel's source-level types.

3.2.4 Validating Function Pointers

Once the monitor has located a particular function pointer, it must validate whether the target of that pointer is consistent with the kernel's CFG. We have identified four possible approximations for determining consistency, which we list from least to most precise:

- *Valid code region.* In this approximation, the monitor simply requires that all pointers target some portion of valid code, i.e., the set V calculated during the text validation phase (Section 3.1). The performed check is a range comparison within the (small) list of valid ranges.

- *Valid function.* In this approximation, the monitor maintains a list of valid kernel function start addresses for code regions in the set V, and requires that all function pointers target one of these addresses. The performed check is set membership in the large set of allowable function pointers.

- *Valid function type.* This approximation narrows the set of functions that a given pointer can target by maintaining a set of valid function addresses for each function pointer type. The performed check is, first, a lookup for the correct set and, second, a set membership check.

- *Valid points-to set.* This approximation utilizes a static or dynamic points-to analysis for each function pointer in the kernel. At runtime, the monitor requires that any encountered function pointer must target one of the functions in its corresponding points-to set. The performed check is the same as in the valid function type case, but the number of sets is likely to be much larger (one per data-structure member, rather than one per-type).

In theory, these approximations could fail to detect an attack that is able to persistently reuse some or all of the kernel's existing functionality (reminiscent of "jump-to-libc"-style attacks [31]). However, we believe that the above approximations will defend against many such attacks because of the difficulty of reusing complete functions for the wrong purpose. Many modern jump-to-libc attacks work by jumping into the middle of code or data that would not be considered valid by our approximations. We have implemented the first two approximations and found that both were sufficient to detect the control-modifying attacks in our corpus.

3.2.5 Monitor Timing

A natural approach to monitoring is to periodically pause the target VM long enough for the monitor process to traverse and validate the target kernel's state. This pause can be disruptive, however; in our benchmarks we have seen the traversal take as long as two seconds. Instead, we could reduce the pause to be just long enough to copy the kernel's memory to the monitor process where it can be traversed asynchronously. Unfortunately, to do this requires allocating a substantial amount of memory to the monitor process that we would prefer to allocate to the target; the Linux kernel, for example could occupy up to 1 GB of memory.

To avoid these problems, we allow the monitor process to traverse the target kernel's heap in parallel with the target VM's execution. While better performing, this approach could result in false positives because the monitor may view the kernel's memory inconsistently. For example, the monitor could queue a pointer whose memory is freed by the kernel before the monitor processes it, and thus the monitor will examine stale data. As a result, it may incorrectly conclude that a bogus bit pattern is a valid pointer and follow it, and/or that a bogus bit pattern is an invalid function pointer and complain about it. In the worst case, the monitor could end up traversing stale data indefinitely. Though perhaps less likely, the same problems could arise even from a snapshot taken at a single moment in time, since the paused kernel may be in the middle of a code sequence it assumes will be atomic. For example, it could be in the middle of adding, removing, or initializing an element in a list, and thus the monitor could end up traversing uninitialized or otherwise invalid pointers.

Our monitor implements three safeguards to mitigate problems due to traversing inconsistent states. First, before following a pointer (or validating a function pointer), the monitor confirms that the pointer targets a valid kernel address by consulting the target's page tables. This prevents reading nonsensical or non-kernel data. Second, the monitor places an upper limit on the number of objects traversed to ensure termination. For our experiments (Section 4), we utilized an upper limit of 2^{20} objects; the limit was never reached during testing, but has been left in place for safety. For our tests, at most 155,328 objects were encountered on any

single pass. Finally, when validating function pointers, our monitor requires the same potential violation (determined by the violating function pointer's address and the address it points to) to be detected in two consecutive monitor runs before raising an alarm. When running at large intervals (currently three seconds or greater), the second "validation" run is commenced within three seconds, rather than waiting for the entire monitor period to expire. This narrows the window between detection and notification, while still allowing the performance tuning to remain in place. We experienced no false positives during any of our experiments using these simple techniques.

4. EXPERIMENTS AND RESULTS

In this section, we present the results of a series of experiments performed using our VMM-based SBCFI monitor. We used the Xen and VMware Workstation VMMs [5, 34] for our test platforms because they could run unmodified kernels on which we could install a large percentage of the collected attacks.

We found that SBCFI could detect all of the control-flow modifying attacks that we were able to install on our target system. We also measured the overhead that monitoring imposed on the system. We found that the SBCFI monitor itself incurs unnoticeable overhead for both a typical web server workload and for CPU-intensive workloads when operating at 1 second intervals on a separate processor from the target kernel. The VMMs added up to 6.6% overhead for two CPU configurations and up to 9.3% for 1 CPU configurations (with or without the monitor), though much less on average. The VMM overhead could potentially be reduced by using a more efficient VMM or by implementing optimizations specific to control-flow monitoring.

4.1 Detection

To demonstrate the effectiveness of SBCFI at detecting kernel attacks, we collected as many publicly available rootkits as we could and tested them on our target platform. Of the 25 that we acquired, we were able to install 18 in our virtual test infrastructure. The remainder either did not support our test kernel version or would not install in a virtualized environment. We installed the rootkits using one of two mechanisms – malicious LKM loading or injection via `/dev/kmem`, a virtual device object that gives direct access to kernel virtual memory.

The protected kernel for all tests was the Linux 2.4.18-3 kernel, the default installation kernel for RedHat Linux 7.3. No modifications were made to this kernel for testing. While generating our monitor for this kernel, our source-code analysis tools extracted 1049 types and 22182 global names, 5105 of which were functions. Based on the calculated type graph, we identified 400 globals as containing function pointers and 260 as viable starting points for reaching function pointers somewhere in memory.

As shown in Table 1, we successfully detected all of the attacks that make persistent modifications to the kernel's control-flow. For loaded LKMs, our monitor detected both the existence of an untrusted module and any function pointer references to that module. Direct-injection attacks, and those module attacks that remove themselves from the module list for stealth, were detected when a persistent pointer targeted the injected code. On several occasions, our monitor also detected the transient changes introduced as part of the direct injection attacks (in addition to their persistent changes). The only attack not detected, `hp`, makes no persistent control-flow modifications. We encountered no false positives during any of our benchmarks or detection tests.

Target Hardware Configuration

Machine Type:	Dell Precision 490 Workstation
Processor:	Intel Xeon Quad-core X5355, 2.66GHz
RAM:	4GB
Storage:	160GB SATA
Networking:	Broadcom NetXtreme BCM5752 Gigabit

Target Software Configuration

Version:	Xen-3.1.0 or VMware Workstation 6.0.1-55017
Host OS:	Debian Etch, full installation Linux 2.6.18-5-686 kernel
Guest OS:	RedHat Linux 7.3 full installation
Guest memory:	1200M

Table 2: SBCFI test platform summary.

At runtime, our monitor visited, on average, 82,097 objects and validated 39,278 function pointers per iteration. The average runtime per iteration was 624 milliseconds for our two-CPU configuration and 1.78 seconds for one CPU. We found that these statistics varied greatly depending on system load, but they never exceeded 155,328 nodes visited and 58,803 function pointers. Maximum traversal time was as long as 2.0 seconds for the two-CPU configuration and as long as 4.8 seconds for one CPU. In general, the number of objects visited increased with system uptime.

4.2 Performance

To evaluate the performance impact of SBCFI monitoring, we measured the performance of the target VMM using subsets of the SPECweb2005 and SPECCPU2006 benchmark suites [32]. We conducted these benchmarks for each of the Xen and VMware platforms. Additionally, we ran each test on each platform in both one-CPU and two-CPU configurations. In the one-CPU configuration, all virtual machines and the VMM shared a single processor. In the two-CPU configuration, the monitor's VM and the target VM could utilize separate processors simultaneously.

For the web benchmark tests, we ran the SPECweb2005 Ecommerce workload, which simulates dynamic and static content delivery over both HTTP and HTTPS. We used two additional systems to act as a request-generating client and a back-end (database) workload simulator (BeSim) respectively. All SPECweb2005 requirements were met during the Ecommerce tests. Details of the target system can be found in Table 2 and details of the client and BeSim systems can be found in Table 3.

In addition to the web benchmarks, we tested the integer workloads (referred to collectively as SPECINT2006) of the SPECCPU2006 benchmark suite on the target system. Like the web benchmarks, we ran these tests both with and without the monitor for one-CPU and two-CPU configurations. The same target machine, summarized in Table 2, was used for these tests.

4.2.1 Results

Table 4 shows the median *Total Weighted Aggregate Byte*

BeSim Hardware Configuration

Machine Type:	Lenovo Thinkpad T60
Processor:	Intel CoreDuo T2500, 2GHz
RAM:	1.5GB
Storage:	80GB IDE
Networking:	Built-in Intel 82573L Gigabit

BeSim Software Configuration

OS:	Gentoo Linux Base 1.12.9
	Linux 2.6.20.1 kernel

Client Hardware Configuration

Machine Type:	Dell Dimension 4700
Processor:	Intel Pentium 4, 2.8GHz
RAM:	1GB
Storage:	40GB IDE
Networking:	Netgear RTL-8169 Gigabit PCI

Client Software Configuration

OS:	Debian Etch, full installation
	Linux 2.6.18-5-686 kernel

Table 3: SPECweb2005 test platform summary.

Configuration	Total ABR	% Native	% VMM
Native	94547.6	0%	—
Xen 3.1.0	94900.1	-0.4%	0%
SBCFI (Cont.)	95018.3	-0.5%	-0.1%
SBCFI (1s)	94941.7	-0.4%	0%
SBCFI (3s)	94351.3	0.2%	0.6%
VMware 6.0.1	94445.5	0.1%	0%
SBCFI (Cont.)	93459.1	1.2%	1.0%
SBCFI (1s)	94457.7	0.1%	0%
SBCFI (3s)	94155.4	0.4%	0.3%

Table 4: SPECweb2005 2CPU ECommerce results.

Rate (ABR) for three SPECweb2005 Ecommerce test runs, performed for each of the listed configurations. There are three types of configurations shown. The first row presents measurements for the physical machine, without the VMM installed. In this configuration, we installed the base Linux distribution (Debian Etch) on the bare hardware and gave it full access to two of the four processors and all 4GB of RAM. This configuration provides a useful basis for determining the overhead incurred by the VMM itself. The second row presents the same measurements for the Xen guest machine, without any monitor. The following three rows describe tests in which the measured Xen guest was being monitored by our SBCFI implementation, which was configured to run asynchronously at various periods (continuously, every second, or every three seconds). Rows six through nine describe the same tests as rows two through five, except that VMware Workstation was used instead of the Xen VMM. The second column provides the median ABR for three runs in each configuration. The third and fourth columns show the incurred overhead (as a percentage) in comparison with the bare hardware (row one) and VMM only (row two or row five) configurations respectively.

Table 5 is formatted the same as Table 4, except that the data reflects a one-CPU system configuration. Because the SBCFI monitor and target VMM were required to share a processor, we did not run the SBCFI monitor continuously in this configuration. Instead, we ran the SBCFI monitor every 500ms (rows three and seven).

Configuration	Total ABR	% Native	% VMM
Native	94035.1	0%	—
Xen 3.1.0	92892.0	1.2%	0%
SBCFI (.5s)	92412.6	1.7%	0.5%
SBCFI (1s)	92841.8	1.3%	0.1%
SBCFI (3s)	92982.9	1.1%	-0.1%
VMware 6.0.1	93418.1	0.7%	0%
SBCFI (.5s)	93250.3	0.8%	0.2%
SBCFI (1s)	93347.1	0.7%	0.1%
SBCFI (3s)	93471.5	0.6%	-0.1%

Table 5: SPECweb2005 1CPU ECommerce results.

Configuration	Time (s)	% Native	% VMM
Native	9956	0%	—
Xen 3.1.0	10610	6.6%	0%
SBCFI (Cont.)	10796	8.4%	1.8%
SBCFI (1s)	10687	7.3%	0.7%
SBCFI (3s)	10634	6.8%	0.2%
VMware 6.0.1	10543	5.9%	0%
SBCFI (Cont.)	10931	9.8%	3.7%
SBCFI (1s)	10607	6.5%	0.6%
SBCFI (3s)	10585	6.3%	0.4%

Table 6: SPECINT2006 2CPU median run times.

From these results we make two observations. First, the overhead incurred by running the VMM without the monitor is small – less than 1% for VMware and just over 1% for Xen when sharing a single processor. For the two-CPU configuration, VMware showed a very small penalty (.1%) and Xen actually performed slightly better (.4%) than the native Linux distribution. Second, SBCFI adds minimal (1% or less) in all configurations; in some cases, performance actually improves slightly. At this time, we cannot directly account for the performance improvements (virtualized over native or with-monitor over without-monitor), but given the complexity of modern machines and the susceptibility of performance to change drastically following minor changes, we are not surprised. Possible contributing factors include the interaction of multiple schedulers (VMM and guest kernel) and the system's complex memory hierarchy.

Table 6 shows the results of the SPECINT2006 tests for the two-CPU configuration. The results listed in the table are organized similar to the web benchmark results. The first column indicates which configuration was under test. The second column shows the time in seconds for the median of three runs of the twelve SPECINT2006 workloads. The third and fourth columns show overhead relative to the raw hardware and VMM-only configurations respectively. Table 7 is similar to Table 6, but shows the results for our one-CPU setup.

The SPECINT2006 experiments show that the majority of the overhead is the result of the VMM and not SBCFI, except for the one-CPU Xen configuration, where SBCFI

Configuration	Time (s)	% Native	% VMM
Native	9954	0%	—
Xen 3.1.0	10883	9.3%	0%
SBCFI (5s)	12297	23.6%	13.0%
SBCFI (10s)	11649	17.0%	7.0%
SBCFI (30s)	11161	12.1%	2.6%
SBCFI (60s)	11024	10.8%	1.3%
SBCFI (90s)	10973	10.2%	0.8%
VMware 6.0.1	10782	8.3%	0%
SBCFI (5s)	11127	11.8%	3.2%
SBCFI (10s)	10987	10.4%	1.9%
SBCFI (20s)	10860	9.1%	0.7%

Table 7: SPECINT2006 1CPU median run times.

imposes up to 13% penalty when operated at 5-second intervals. In all cases, SBCFI itself imposes a tunable penalty, trading off precision for performance, on top of the VMM, with unnoticeable overhead when monitoring at 1-second intervals for the two-CPU configuration (using either VMM) and 90-second or 20-second intervals for the one-CPU configuration when using Xen or VMware respectively.

4.2.2 Discussion

Our experiments show that the overhead of SBCFI on its own is quite small, and that the primary cause of overhead is due to the VMM itself, particularly when multiple processors are available. We do not believe this reflects poorly on SBCFI itself for two reasons.

First, the VMM overhead is a function of the VMM we used, not of VMM technology in general. As described previously, we chose VMMs and configurations in our test platform to maximize threat testing rather than performance. We ran Xen in its fully virtualized (as opposed to paravirtualized) mode to support an unmodified kernel on which we could use unmodified attacks. Xen can achieve better performance for fully virtualized hosts with the use of special drivers (around 2% of native on average [36]), and there is nothing that precludes SBCFI monitoring of either paravirtualized or fully virtualized hosts using these drivers. Our VMware implementation performance could be improved yet further by using VMware's high-performance ESX Server (if provided with the appropriate APIs). The increasing deployment of high-performance virtualization solutions [4] provides further evidence that VMMs can have reasonable performance when compared to raw hardware.

Second, SBCFI could be implemented without a VMM, e.g., as a separate card that accesses memory directly (in which case, however, the monitor would not have access to the machine's registers). Petroni et al.'s Copilot system [23] performs integrity monitoring in this way, and for comparable monitoring rates they induce similarly low overheads on the target kernel.

In short, our experiments show that SBCFI is effective and practical, detecting all of the kernel attacks we could install on our platform while imposing minimal impact on top of the VMM.

5. RELATED WORK

There is a growing body of work in the area of kernel integrity monitoring related to monitoring mechanisms, detection algorithms, and advanced threats. Garfinkel et al. first proposed using a virtual machine monitor to implement a protected system integrity monitor, including invariant-based kernel protection [14]. Their Livewire system is capable of verifying the kernel's text regions, looking for specific types of data attacks that can be detected by querying the system at different levels, and verifying static function pointer tables (e.g., the system call table). Our implementation borrows their VMM-based mechanism but enforces a far more comprehensive kernel integrity policy with SBCFI.

Zhang et al. [37], followed by Petroni et al. [23] propose to deploy a kernel monitor on a trusted piece of hardware, such as a PCI add-in card. While add-in hardware does not generally have full access to the CPU's state (most notably its registers), it enjoys improved performance over VMM-based monitoring and can be deployed in nearly any system. While we have not done so, SBCFI could be implemented using trusted hardware.

Seshadri et al. propose Pioneer, a technique for running trusted software on an untrusted system in a verifiable way ("verifiable code execution") [28]. Unlike attestation-based techniques [18, 13, 29, 26, 27, 30], which measure code and data at load time, Pioneer is capable of running arbitrary procedures in a trusted manner. The authors demonstrated this capability by implementing a runtime kernel integrity monitor and proving that it could detect real-world attacks by verifying immutable text and data. While currently impractical for operational systems (e.g., it requires disabling interrupts while running the verification procedure) verifiable code execution could be a viable future platform for enforcing SBCFI.

Grizzard proposes to use a VMM to monitor the kernel's execution and validate its control flow [15]. The system works by rewriting the target kernel to trap all dynamic branches into the VMM before they are performed. The control-flow monitor then verifies that the branch is consistent with the kernel's CFG, determined through prior training runs. Though he does not specifically describe CFI, Grizzard's implementation effectively enforces CFI for the OS kernel. The clear advantage is that, by enforcing true CFI, all violations of the CFG, even those that are transient, can be detected. The primary disadvantage of this approach is the incurred overhead and the challenges facing the reduction of that overhead. For the lmbench synthetic benchmark, Grizzard reports an average of 30% performance penalty (worst case 74%) on top of the VMM's existing overhead. Another challenge is how to handle new kernel modules, particularly since the CFG is obtained via training.

Litty and Lie also propose a VMM-based system, called Manitou, for validating the executing code of both user applications and the kernel within a guest VM [20]. The VMM maintains a list of cryptographic hashes of the in-memory representations of application and kernel-level code pages that may be run within the VM. By default, the VMM sets all guest VM pages as non-executable, using hardware support only recently available on the x86. Attempts to execute such pages fault into the VMM, which will set the execute bit and permit execution if the offending page matches one in its trusted list (further logic is used to prevent subsequent modifications).

Manitou's use of execute bits prevents unauthorized code from ever executing—a stronger guarantee than SBCFI. On the other hand, the approach, as proposed, only enforces

that valid code pages are executed, not that execution proceeds according to a valid CFG; this is similar to the first validation option presented in Section 3.2.4, and will miss at least some "jump-to-libc"-style attacks, even persistent ones. Manitou's reliance on page faulting requires a VMM-based implementation; SBCFI can be implemented using a PCI card or other external device for greater tamper-resistance of the monitor. As a possible inhibitor to practical deployment, Manitou's use of execute bits prevent its operating in an "audit only" mode in which execution may proceed despite monitor warnings. This facilitates a trivial denial of service by an attacker that is able to modify on-disk executables. Finally, overhead is incurred on every change to executable content, such as during normal OS demand-paging. Indeed, the approach has yet to be thoroughly studied on real systems, and so the details of a practical implementation and its overhead are as yet unknown.

The commercial and applied security communities are also interested in kernel integrity and rootkit detection (examples include [19, 8, 12, 25]). Existing tools look for specific signs of compromise within the system by verifying invariants of low-level data structures, by observing system API behavior for inconsistencies, or both. Some tools enforce control-flow properties for a small subset of kernel text and data. SBCFI can be viewed as a generalization of these techniques that provides far more complete protection. For example, a number of recent tools verify specific kernel function pointers, such as those found in well-known jump tables and kernel subsystems (e.g., the virtual file system) [19]. As a result, improvising attackers have turned to making modifications deeper within the system, finding function pointers and code not verified by current tools [2]. SBCFI removes this avenue from the attacker by checking a much larger, systematically-determined set of function pointers. We conclude that SBCFI is the most effective kernel integrity monitor to date: it is practical and efficient enough to use on real systems, but detects far more attacks without a priori knowledge of the specific changes an attacker might make.

As an alternative or complement to dynamic monitoring, the kernel's vulnerability to attack can be reduced by using static analysis and special compilation. Deputy [9, 38] is a compiler and annotation system for C with which programmers can enforce partial memory safety (a garbage collector is required for complete protection), thereby ensuring attackers cannot overrun buffers or perform similar attacks to inject illicit functionality into the kernel in the first place. Related systems include Cyclone [17] and CCured [22], but both of these require representation-modifying compilation (e.g., to introduce so-called "fat" pointers). All three systems have been used to write kernel components [38, 10, 33], but to date none has proven practical for a complete kernel. We observe that some of Deputy's annotations would be useful in our SBCFI implementation, though we do not require annotations on or within functions, and would not require special compilation.

While the above systems operate at the source level, the program's binary can be rewritten to insert integrity-protecting checks that occur during execution. Abadi et al. implement CFI enforcement in this way [1]. Their implementation is an instance of a general technique called in-line reference monitoring (IRM) that can be used to enforce a range of policies [11, 35]. IRMs typically rely on control-flow and other restrictions to guarantee that their checks are not circumvented (e.g., jumping to code that executes immediately after the check). These restrictions can be difficult to enforce within the kernel, as described in Section 2.1.

6. CONCLUSION

This paper has presented a new approach to dynamically monitoring operating system kernel integrity based on a property called *state-based control-flow integrity* (SBCFI), an approximation of Abadi et al.'s Control-Flow Integrity property. Violations of SBCFI indicate a persistent, unauthorized modification of the kernel's control-flow graph, which we have shown to be common in kernel attacks. We have implemented a virtual machine monitor-based SBCFI monitor for the Linux kernel. We demonstrated experimentally that our monitor can successfully detect a substantial number of real kernel attacks (17 of the 18 we injected in our test platform) with no false positives and low overhead, on average as compared to the VMM. We believe our SBCFI monitor is the most effective kernel integrity monitor to appear in the literature to date. In future research, we plan to explore other properties that consider attacker incentives, to target rootkits that SBCFI would fail to detect.

Acknowledgments

We thank Jeff Foster, Tim Fraser, Pavlos Papageorgiou, Boniface Hicks, and Trevor Jim for commenting on a draft of this paper, and the anonymous referees for their insightful feedback. We also thank the VMware benchmark team for their valuable and quick feedback. This work was supported in part by NSF grant CCF-0524036.

7. REFERENCES

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

[2] Aditya Kapoor and Ahmed Sallam. Rootkits Part 2: A Technical Primer. `http://www.mcafee.com/us/local_content/white_papers/wp_rootkits_0407.pdf`, 2007.

[3] A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *2007 IEEE Symposium on Security and Privacy*, May 2007.

[4] E. Bangeman. Gartner: virtualization to rule server room by 2010. `http://arstechnica.com/news.ars/post/20070508-gartner-virtualization-to-rule-server-room-by-2010.html`, May 2007.

[5] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the USENIX Security Symposium*, August 2005.

[8] B. Cogswell and M. Russinovich. Microsoft rootkitrevealer. `http://www.microsoft.com/technet/`

sysinternals/utilities/RootkitRevealer.mspx, 2007.

[9] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *Proceedings of the European Symposium on Programming (ESOP)*, Apr. 2007.

[10] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the Real World. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[11] Ú. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: a Retrospective. In *Proceedings of the Workshop on New Security Paradigms*, 2000.

[12] F-Secure. F-secure blacklight. `http://www.f-secure.com/blacklight/blacklight.html`, 2007.

[13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[14] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, February 2003.

[15] J. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

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

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[18] R. Kennell and L. H. Jamieson. Estabilishing the Genuity of Remote Computer Systems. In *Proceedings of the USENIX Security Symposium*, August 2003.

[19] T. Klein. Rootkit profiler LX. `http://www.trapkit.de/research/rkprofiler/rkplx/rkplx.html`, 2007.

[20] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In *Proceedings of the Workshop on Architectural and system support for improving software dependability (ASID)*, 2006.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, Apr. 2002.

[22] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2002.

[23] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the USENIX Security Symposium*, August 2004.

[24] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the USENIX Security Symposium*, August 2006.

[25] H. Saidi. Guarded models for intrusion detection. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2007.

[26] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based Policy Enforcement for Remote Access. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2004.

[27] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the USENIX Security Symposium*, August 2004.

[28] 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 *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[29] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[30] E. Shi, A. Perrig, and L. V. Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

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

[32] SPEC. Standard Performance Evaluation Corporation. `http://www.spec.org`, 2007.

[33] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Science of Computer Programming (SCP)*, 62(2):122–144, Oct. 2006. Special issue on memory management.

[34] VMware, Inc. VMware Workstation. `http://www.vmware.com`, 2007.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, December 1993.

[36] XenSource, Inc. A Performance Comparison of Commercial Hypervisors. `http://www.xensource.com/files/hypervisor_performance_comparison_1_0_5_with_esx-data.pdf`, 2007.

[37] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the ACM SIGOPS European Workshop*, September 2002.

[38] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, Nov. 2006.