Comprehensive and Efficient Protection of Kernel Control Data

Jinku Li, Zhi Wang, *Member, IEEE*, Tyler Bletsch, Deepa Srinivasan, Michael Grace, *Student Member, IEEE*, and Xuxian Jiang, *Member, IEEE*

***Abstract*—Protecting kernel control data (e.g., function pointers and return addresses) has been a serious issue plaguing rootkit defenders. In particular, rootkit authors only need to compromise one piece of control data to launch their attacks, while defenders need to protect thousands of such values widely scattered across kernel memory space. Worse, some of this data (e.g., return addresses) is volatile and can be dynamically generated at run time. Existing solutions, however, offer either incomplete protection or excessive performance overhead. To overcome these limitations, we present indexed hooks, a scheme that greatly facilitates kernel control-flow enforcement by thoroughly transforming and restricting kernel control data to take only legal jump targets (allowed by the kernel's control-flow graph). By doing so, we can severely limit the attackers' possibility of exploiting them as an infection vector to launch rootkit attacks. To validate our approach, we have developed a compiler-based prototype that implements this technique in the FreeBSD 8.0 kernel, transforming 49 025 control transfer instructions (∼7.25% of the code base) to use indexed hooks instead of direct pointers. Our evaluation results indicate that our approach is generic, effective, and can be implemented on commodity hardware with a low performance overhead (<5% based on benchmarks).**

***Index Terms*—Intrusion prevention and tolerance, software, system design and implementation.**

I. Introduction

MODERN operating systems (OSs) are vulnerable to various types of attacks. In particular, kernel-level rootkits have been a growing threat [1], [2] due to their stealthy nature and omnipotent residence on a compromised system. Specifically, these rootkits typically run at the highest privilege of the system (e.g., the *root* privilege in a UNIX system) and effectively hide themselves inside the system. Facilitated by their stealthy presence, they are often employed to perform various nefarious activities, including disabling defense mechanisms, stealing sensitive personal information, opening remotely controllable back doors, etc.

To address this security threat, there is a need to thoroughly safeguard the integrity of kernel code and data. Note that static kernel code and static kernel data are relatively straightforward to protect due to their read-only nature and well-defined locations in kernel memory. A number of systems have been accordingly proposed and implemented. As an example, SecVisor [3] is a hypervisor-based system that prevents guest kernel code from being comprised by strictly enforcing the $W \oplus X$ [4] property. NICKLE [5] is another example that proposes a separate code memory to store legitimate kernel code. This code memory is used to guarantee the kernel code integrity by ensuring that every instruction running at the kernel mode will be fetched *only* from it.

Unfortunately, dynamic kernel data is much harder to protect due to its unpredictable memory location and volatile nature. Yet, this data can still be potentially exploited to launch similar rootkit attacks. For example, return-oriented programming [6], [7], or more specifically return-oriented rootkits (RORs) [8], bypass existing kernel code integrity protection while still performing Turing-complete malicious computation. Specifically, by hijacking a function pointer or a return address, a return-oriented rootkit redirects execution to its own gadgets, consisting only of legitimate kernel code, to unfold its payload with unfettered access to kernel memory.

In this paper, we focus on the protection of an important type of kernel data: kernel control data. In particular, kernel data is considered control data if it is loaded into the processor's program counter at some point in kernel execution. Accordingly, there are two main types of kernel control data: *function pointers* and *return addresses*. For ease of presentation, we use the term kernel control data and kernel hooks interchangeably.

We note that a number of systems have been proposed for user-level control data protection. For example, Program Shepherding [9] uses a dynamic machine-code translation technique to constrain the control-flow transfer of a running program. However, its complexity may affect its trustworthiness and complicate its adoption [10]. Control-flow integrity (CFI) [10] instead instruments the control-flow transfer instructions of a program to ensure that the running instance can always jump to the right instruction when a control-flow transfer occurs. However, due to additional unique challenges and complexities in the kernel space, its effectiveness in providing comprehensive

Manuscript received July 08, 2010; revised April 29, 2011; accepted May 27, 2011. Date of publication June 16, 2011; date of current version November 18, 2011. This work was supported in part by the U.S. Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099, by the U.S. National Science Foundation (NSF) under Grant 0852131, Grant 0855297, Grant 0855036, and Grant 0952640. It was also supported in part by the Fundamental Research Funds for the Central Universities of China under Grant K50511030002 and the National Natural Science Foundation of China under Grant 61003300. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. R. Sekar.

J. Li was with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA. He is now with the School of Computer Science and Technology, Xidian University, Xi'an 710071, China (e-mail: jkli@xidian.edu.cn).

Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA (e-mail: zhi_wang@ncsu.edu; tkbletsc@ncsu.edu; dsriniv@ncsu.edu; mcgrace@ncsu.edu; xuxian_jiang@ncsu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2011.2159712

kernel control data protection still remains to be shown. Some obvious challenges include the presence of a large number of dynamic function pointers in contemporary OS kernels [11] as well as the support for loadable kernel modules (LKMs), which dynamically spans or shrinks the runtime kernel control-flow graph.

From another perspective, in order to protect kernel hooks, an intuitive approach is to systematically identify all of them and apply hardware-based page-level protection to trap all writes to those memory pages containing hooks. This approach is appropriate if there are only a few hooks to protect (e.g., in secure active monitoring [12]). However, it cannot be applied directly to protect a large number of widely scattered hooks in the OS kernel. The reason is that if hooks are widely scattered in memory and potentially coexist with other noncontrol data in the same pages, a direct application of page-level protection will lead to frequent, expensive page faults [11], thus leading to excessive performance overhead. HookSafe [11] recognizes this challenge as the *protection granularity gap* (where hook protection requires byte-level granularity while hardware provides only page-level granularity) and proposes a solution that relocates kernel hooks to a centralized location and then applies the hardware's page-level protection. To allow seamless hook access, a thin hook indirection layer is introduced to regulate accesses to them, hence avoiding unnecessary overhead caused by trapping writes to irrelevant data. However, HookSafe still suffers from a few significant limitations. For example, it only protects a given set of function pointers. The system itself is unable to exhaustively discover all hooks for protection. Also, it by design still leaves another type of kernel control data, return addresses, unattended. As mentioned earlier, these addresses can be equivalently hijacked for rootkit purposes.

To bridge the protection granularity gap without the above limitations, we propose a compiler-based approach to comprehensively and efficiently safeguard kernel control data. The central idea of our approach is the notion of *indexed hooks* to facilitate kernel control-flow enforcement. Specifically, our approach proactively transforms kernel control data into indexes of read-only jump tables, which contain only legitimate jump targets allowed by the kernel's control-flow graph (CFG). As a result, we can effectively prevent attackers from overwriting the kernel control data with arbitrary pointers, hence preventing them from being misused to launch rootkit attacks. Our scheme is motivated by HyperSafe [13], one of our prior works that aims to enforce CFI for small, static, and single-threaded Type-I hypervisors (e.g., BitVisor [14]). In this work, we take a step further and apply this scheme to the protection of commodity OS kernels (more specifically, the kernel hooks), which involves addressing additional nontrivial challenges that are unique to the OS kernel context. For example, commodity OS kernels by design support multitasking (instead of single-threaded execution) and are more dynamic (with their built-in LKM support). These additional challenges are naturally reflected in our system prototype (Section III).

Our scheme essentially transforms the original control data into a new form that will take legal values *only*. This is in contrast with the existing form of control data, which is blindly trusted to contain valid jump targets and has been widely exploited by attackers to compromise and redirect control to anywhere they want. Our scheme handles the two types of kernel control data differently. Specifically, to transform function pointers, our first step is to systematically locate all of them. For that, we notice that each function pointer will be eventually invoked by an *indirect call/jmp* instruction in the machine code. As such, we can leverage the compiler to identify all indirect call/jmp instructions at the intermediate representation (IR) level. After that, we then locate and instrument related control data into indexes so that they can be properly redirected to their destinations contained in a jump table or function table. For each *indirect call/jmp*, its function table contains all the function entry points it may enter according to the CFG. Note that for an indirect call/jmp, the jump target will be preloaded into a register or a memory location by a previous instruction (e.g., *mov* or *lea*). Fortunately, at the IR level, such a load instruction will have, as one of its operands, the function symbol. With that, we can then extend the compiler to recognize and replace it with an index into the corresponding function table in the first place. Later on, when an indirect call is made, its original destination can be naturally obtained by performing a lookup on its function table. We note that the function tables are read-only and thus can be protected with the hardware-based page-level protection.

For another type of control data, return addresses, the transformation process is different, as return addresses are dynamically generated. More specifically, these return addresses are pushed onto the stack by *call* instructions, either direct or indirect, and popped off by *ret* instructions. As a result, a return address should point to the next instruction right after the corresponding *call*. Consequently, we can precompute all valid targets for return instructions and store them into another type of jump table or return table. For each *ret*, its return table contains all the return addresses it may return to according to the CFG. With that, we can leverage the IR to instrument the related call/ret instructions. Also note that the return tables are read-only, implying that we can align them at the page boundary and apply the same hardware-based page-level protection to efficiently protect them.

To validate our approach, we have implemented a proof-of-concept prototype based on the open-source LLVM compiler [15]. As a compiler-based approach, our system requires recompiling the OS kernel source code but is capable of safeguarding all control data in the OS kernel. Specifically, we have used our system to recompile a protected version of FreeBSD 8.0/x86-amd64 kernel. The protection is achieved by transforming 49 025 control transfer instructions, which occupy $\sim$7.25% of the entire kernel code base. After the transformation, our system enforces them to take only legal jump targets allowed by the kernel's control-flow graph. In summary, our paper makes the following contributions:

1) We recognize the difficulties in comprehensively protecting kernel control data and accordingly propose indexed hooks as our solution to proactively transform and safeguard it. Our scheme effectively prevents rootkits from using kernel control data as an infection vector.
2) To validate our approach, we have developed a compiler-based prototype. Specifically, our prototype is based on the

open-source LLVM compiler and has been used to compile the FreeBSD 8.0 kernel. The compilation process essentially permeates the legitimate kernel CFG information into every transformed control transfer instruction for enforcement. We have so far successfully applied this approach to enforce FreeBSD kernel control-flow with two different granularities (Section II).

3) We perform a systematic security analysis and conduct a number of synthetic attacks against our system. The experimental results show that our system can prevent various kernel attacks from hijacking control data, including recent return-oriented ones. Our performance measurement indicates our prototype has a low performance overhead ($<5\%$ based on benchmarks).

The rest of the paper is organized as follows. First we describe the design goals, threat model, and our key techniques in Section II. Then we present the implementation details and evaluation results in Sections III and IV, respectively. After that, we discuss possible limitations of our current prototype in Section V and describe related work in Section VI. Finally, we conclude our paper in Section VII.

II. System Design

In order to provide comprehensive and efficient kernel control data protection, we have three main design goals. First, our proposed techniques should be able to recognize and protect all the control data in the OS kernel, including function pointers as well as return addresses. Ideally, we aim to have a unified scheme that effectively defeats recent return-oriented attacks [6]–[8] and goes a step further from current research efforts [3], [5] that enforce kernel code integrity.

Second, our proposed techniques should not require redesigning or dramatically changing the original kernel while still guaranteeing comprehensive control data protection. In particular, it should require minimal or ideally no modification to the modern OS kernel. In other words, our techniques should be generic and portable to other commodity OSs.

Third, the proposed techniques should be able to be efficiently implemented and readily deployable on commodity hardware, i.e., without depending on certain sophisticated hardware support for additional features or reducing performance overhead. Given this, the challenge here is to make sure that our proposed techniques will have a low performance impact.

In this work, we assume that the OS kernel code integrity is guaranteed by a trustworthy hypervisor (e.g., by SecVisor [3] or NICKLE [5]). This assumption is necessary as a trustworthy hypervisor establishes the trusted code base (TCB) of the entire system. Note that there has been wide concern regarding the feasibility of the hypervisor's integrity. Fortunately, more recently, there has been significant progress on reducing the hypervisor code size (e.g., with the Xen domain disaggregation [16]), formally verifying the hypervisor security properties [17], as well as enabling hypervisor self-protection from code injection attacks [13]. In the meantime, we assume the adversary can obtain the highest privilege inside the OS (e.g., the *root* privilege in UNIX) and full access to the system memory space (e.g., through /dev/mem in Linux). Also we assume the presence of OS kernel vulnerabilities that can be exploited to overwrite the control data in kernel memory.

We point out that kernel code integrity is essential to our system. Without that, one simple jump instruction to a special destination address can immediately lead to either the execution of injected code or the misuse of existing code (e.g., in a return-oriented rootkit). Further, if we compare it to the user-level control-flow integrity (CFI) [10], the two assumptions of CFI—NWC (Non-Writable Code) and NXD (Non-Executable Data)—are essentially equivalent to the kernel code integrity in our case.

Based on this threat model, our goal is to prevent the attackers' possibility of exploiting the kernel control data as an infection vector (e.g., as demonstrated by the recent return-oriented rootkits [8]). Accordingly, we propose a solution to transform existing kernel control data into its indexes and further limit the choices from these indexes to legal jump targets only. To do that, we recognize and instrument all the hook-accessing instructions in OS kernel, including those control transfer instructions (*call/jmp/ret*). Next, we describe how indexed hooks can be applied for the protection of function pointers and return addresses in detail, respectively. We will also examine possible exceptions and caveats.

A. Transforming Function Pointers

In this subsection, we describe how indexed hooks can be applied for function pointer protection. A function pointer is used in an indirect *call/jmp* instruction. Direct *call/jmp* instructions encode the target address (as an absolute address or relative offset) in the machine code; indirect ones typically need to preload their jump targets by leveraging a machine register (or a specific memory location). Specifically, the jump target will be loaded by an earlier memory load instruction (e.g., *mov* or *lea*) into a register, which is then "consumed" by the indirect call/jmp instruction. With indexed hooks, this memory load instruction should be recognized and instrumented to load the corresponding index, instead of the actual jump target or destination. The index can then be used as an offset into a corresponding *function table*, which contains pointers to the possible kernel function routines that will be indirectly invoked by an indirect call/jmp instruction. In other words, we essentially replace each function pointer with its own index or, more precisely, its *function table index* during an earlier memory load instruction.

The recognition of such load instructions is greatly facilitated by the fact that the intermediate representation (IR) of these instructions will have the corresponding function symbol as one of their operands when the code is being compiled; we can then extend the compiler to recognize them and perform the instrumentation accordingly. For the function tables, we reserve a page-aligned memory area to contain all recognized jump targets used by these function pointers. The jump targets in the tables are naturally ordered based on the assigned indexes and their bases. (Note that each indirect call/jmp instruction can share or own its function table.) With the function tables, when an (instrumented) indirect call/jmp is made, its original destination can be readily obtained by performing a lookup on a function table. It is important to point out that each function pointer

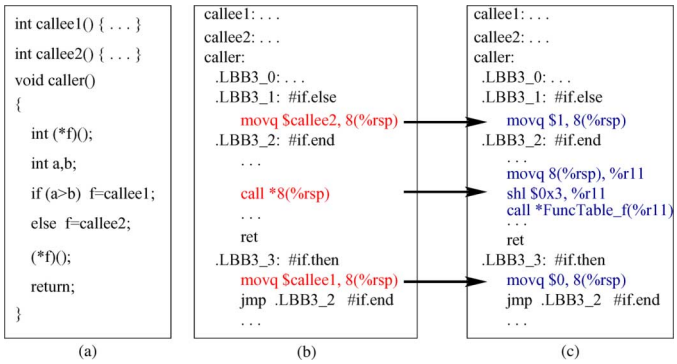


Fig. 1. An example: transforming function pointers. (a) Example.c; (b) original LLVM IR; (c) new IR with indexed hooks.

will point to a valid function in the OS kernel allowed by the CFG. As a result, these function tables can be generated offline and protected at run time with the hardware-based page-level protection. To ensure the protection of *all* function pointers, we note that each function pointer will be eventually invoked by an *indirect call/jmp* instruction in the machine code. Consequently, by instrumenting all of these instructions, we can effectively transform and protect all function pointers, including the volatile ones that are created or removed at runtime as a part of dynamic kernel objects.

Next, we use an example to demonstrate how a function pointer can be translated into its index. In Fig. 1(a), we show a simple C source file *example.c*. In the file, it defines a *caller* function and two *callee* functions, i.e., *callee1* and *callee2*, which will be possibly called by *caller* through a function pointer (named as $f()$) according to the comparison result of variables a and b. In Fig. 1(b), we show the original LLVM backend IR for the file. Particularly, *caller* has been compiled to four basic blocks: *.LBB3_0–.LBB3_3*. In basic blocks *.LBB3_3* and *.LBB3_1*, the addresses of *callee1* and *callee2* are loaded to a memory unit 8(%rsp) by two *movq* instructions. Later on, in the basic block *.LBB3_2*, 8(%rsp) is invoked by an indirect call instruction call *8(%rsp) as the destination operand. From the IR, we can infer this *movq* is actually the previous memory load instruction as discussed earlier. As the first operand of *movq* is the symbol of a callee function (*$callee1* or *$callee2*), we can conveniently recognize it and then replace it with an assigned function table index (e.g., 0 for *callee1* and 1 for *callee2*).

Accordingly, with indexed hooks, we replace call *8(%rsp) with three instructions. [The instrumented result is shown in Fig. 1(c).] The first instruction is a *movq* instruction that loads the content contained in 8(%rsp) to register *r11*. It is important to note that the content of *r11* has been replaced with a function table index instead of the original function address. The second instruction is a *shl* instruction that changes the index in *r11* to the offset of the function table (by multiplying 8). The third instruction is a *call* instruction that uses *r11* as an offset into the corresponding function table (*FuncTable_f*), obtains the actual function address or destination, and transfers the control to it.

Based on the above example, in order to transform function pointers under our scheme, we need to instrument all the *indirect call/jmp* instructions, and those instructions that will preload function pointers with their jump targets. To better understand how our scheme impacts the control flow transfer, we show a comparison between the traditional control flow transfer and our new control flow transfer in Fig. 2. Specifically, in the traditional control flow transfer [Fig. 2(a)], a call pushes a return address onto the stack (*C1:push*) and then transfers the control to its target (*C2:transfer*). In our new control flow transfer [Fig. 2(b)], an instrumented call pushes a return table index on the stack (*C1:push*), locates the jump target from the function table (*C2:locate*), and then transfers control to it (*C3:transfer*). The return table and the related return table index in the figure will be described shortly.

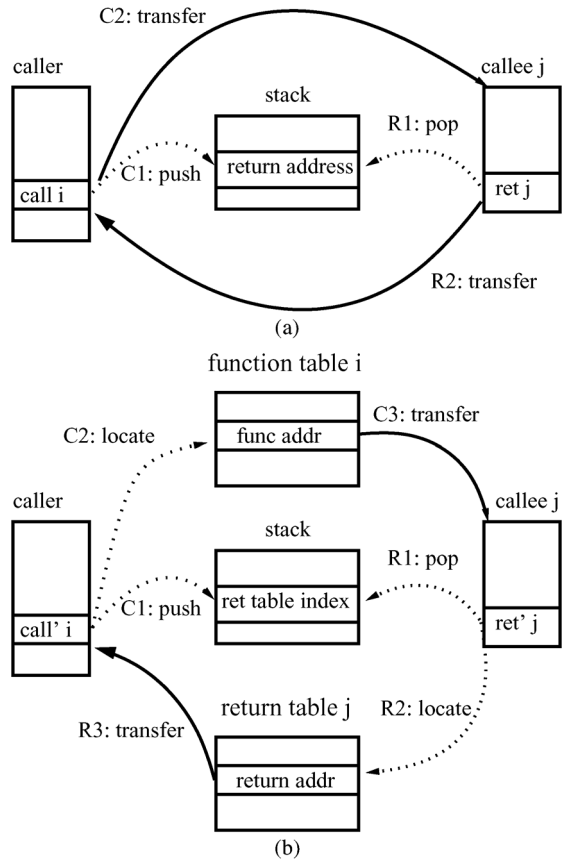


Fig. 2. Traditional indirect call versus new indirect call. (Note: call′/ret′ is the instrumented version of call/ret.) (a) Traditional indirect call; (b) new indirect call.

B. Transforming Return Addresses

Next, we describe how to apply indexed hooks for another type of control data—return addresses. As noted earlier, we cannot handle return addresses in the same way as we instrument function pointers. The reason is that they are volatile and dynamically generated at run time. More specifically, by following the function call convention, when there is a *call* instruction executed, either directly or indirectly, a return address is pushed onto the stack and later it is popped off by a *ret* instruction.

Inspired by the observation that each return address must point to the instruction that immediately follows a *call* instruction, we can precompute all valid return targets and save them in another type of jump table called the *return table*. With the return tables, we then leverage the compiler IR to instrument all the return address-related instructions by replacing the return

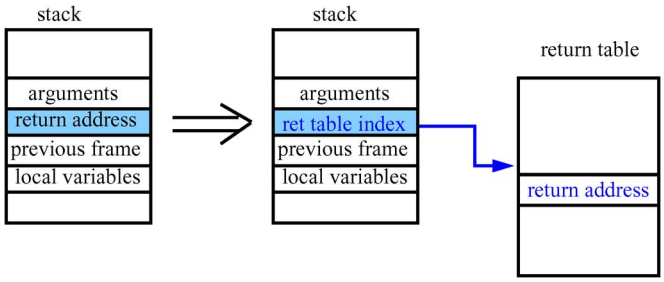


Fig. 3. Transforming return addresses.

address with a *return table index*. Notice that such instrumentation changes the conventional stack frame organization.

To illustrate the change, we show in Fig. 3 the impact on the traditional stack frame. In essence, the traditional return address is replaced with a return table index. In order to locate the return target, we need a return table for each *ret* that contains its legitimate return targets and the index is used to determine which target is being used for this particular *ret* instance. Note that the return table index will be pushed onto the stack by an earlier *call* instruction. Based on the location of the *call* instruction, we can uniquely assign a return table index for each return table so that there is no conflict in the index assignment.

After the return address transformation, the new control flow is shown in Fig. 2(b). Specifically, when a *call* is made, the instrumented execution pushes a return table index onto the stack (not the traditional return address—shown as *C1: push* in the figure). The return table index will point to an entry of its return table and the entry contains the actual return address instead (i.e., the location of next instruction right after the *call*). When executing a *ret*, instead of popping up a return address from the stack, the instrumented version obtains an index (*R1: pop* in the figure), uses it to lookup the actual return address in the corresponding return table (*R2: locate*), then transfers control back to it (*R3: transfer*). Notice that the return table is static and can be naturally protected with the hardware-based page-level protection.

Despite the volatile nature of return addresses, we note that the return address is generated from *call*, either direct or indirect, and consumed by *ret*. As such, by instrumenting all these related instructions, we can effectively protect *all* volatile return addresses by limiting the destinations allowed within these return addresses to the legal values only.

It is important to note that the transformation of function pointers affects indirect call/jmp instructions while the transformation of return addresses affects direct call, indirect call, and ret instructions.[1] However, both transformations need to cooperate with each other due to the new convention of using the return table index in a stack frame. Putting everything together, we obtain the following high-level instrumentation. The detailed algorithm for the instrumentation will be shown in Section III.

- *call dst* $\Rightarrow$ *push $ret_table_index; jmp dst*
- *call *(dst)* $\Rightarrow$ *push $ret_table_index; mov (dst), %reg; shl $0* $\times$ *3, %reg; jmp *FuncTable(%reg)*
- *ret* $\Rightarrow$ *pop %reg; shl $0* $\times$ *3, %reg; jmp *RetTable(%reg).*

[1]There is no need to instrument the direct jmp instructions.

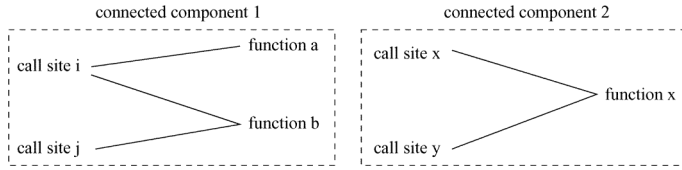


Fig. 4. CFG with two disjoint components.

C. Computing Jump Tables From CFG

After describing how the indirect call/jmp and return instructions are instrumented, next we introduce how to compute their jump targets from the kernel control-flow graph (CFG). At first glance, given a CFG, it might seem straightforward to obtain their jump targets. However, one peculiarity called *destination equivalence* [10] makes it more complicated than it appears. To understand that, we assume there are three indirect call sites, i.e., i, j, and k. Each call site has two legitimate targets: *func1* and *func2* from the call site i; *func2* and *func3* from j; and *func1* and *func3* from k. We may attempt to simply assign {*func1*, *func2*} to function table of call site i, {*func2*, *func3*} to function table of j, and {*func1*, *func3*} to function table of k, respectively. However, such assignment can cause problems as *func2* has been assigned two different index values or offsets in function tables for i and j. To avoid such conflict, we should assign instead {*func1*, *func2*, *error*} to function table i, {*error*, *func2*, *func3*} to function table j, and {*func1*, *error*, *func3*} to function table k, respectively. Here, *error* denotes a special destination to trap an impossible control transfer. In other words, since we can only assign one index to each function or return site, if a function or a return site appears in more than one jump table, it needs to be assigned with the same index among these tables. Specifically, if we view the CFG as an undirected graph, it contains many (possibly small) connected components. Since a *ret* instruction can never return to call sites in two disjoint components of CFG, we only need to guarantee that return indexes are unique among all the return sites in one particular connected component. Fig. 4 shows an example CFG with two connected components. In this CFG, we can have the following index assignment to return sites: *call site i: 1*, *call site j: 2* and *call site x: 1*, *call site y: 2*. Therefore, to assign indexes to return sites in a function's return table, we can collect all the return sites in the connected component of the function in the CFG to one set, and number them increasingly in the order of their addresses. The same approach also applies for the indexes in function tables.

We point out that our scheme is orthogonal to the generation and precision of kernel CFG. That is, the indexed hooks can be applied with input of either coarse-grained or fine-grained CFG. In fact, this is exactly what we have demonstrated in our prototype. For the coarse-grained CFG, we maintain two large jump tables: one function table and one return table. The function table is used by all the indirect call/jmp instructions and it contains the addresses of all the functions that may be indirectly called. The return table instead is used by all the return instructions and includes all the legal return addresses. For the fine-grained CFG, we profile the kernel execution and collect valid jump targets and use these jump targets to refine the coarse-grained CFG (with proper handling of destination equivalence). As a result, multiple function tables and return tables

will be created. We stress that the applicability of our scheme for different granularities is important as we need to handle the large size of modern kernel source code and accommodate some "in-the-wild" practices such as pervasive use of inline assembly and *void* $*$ pointers, which make existing points-to analysis algorithms hard to be applied. Such applicability will also help our scheme to directly benefit from any advance the community is making in improving the scalability and precision of point-to analysis for OS kernel code [18].

To summarize, as a unified scheme applicable for the protection of both function pointers and return addresses, our approach effectively limits jump targets allowed within the control data to the legal values allowed by the CFG. The downside, however, is the overhead in initializing and maintaining the jump tables, and because of them, introducing an extra memory access in the new instrumented *indirect call/jmp* and *ret* instructions. Fortunately, given the static nature of OS kernel text, we can populate all entries in these jump tables offline. Also, the tables are static and thus can be marked as *read-only*. Although our scheme introduces one more memory access in the new instrumented instructions, our experimental results (Section IV) show that the performance overhead is low ($<5\%$ based on benchmarks), likely because these tables are relatively small and have a cache-friendly access pattern.

D. Other Control Data Protection

In addition to function pointers and return addresses, there are other certain context data that could be similarly used as control data. Not surprisingly, they are mainly involved with the OS interface to hardware. For example, hardware registers such as GDTR, IDTR, SYSENTER, and DR0-DR7 contain system-wide configuration information that will be directly used or invoked by hardware. Therefore, they also need to be protected. To do that, we leverage hardware-based virtualization support to intercept and validate any write attempts to these registers. Also, related to the hardware register protection, there are two tables, i.e., GDT and IDT, which contain critical system data structures and their contents must be protected as well. These tables, once initialized, can be marked read-only, thus we can protect them by using the same hardware-based page-level protection.

There is another subtle issue related to the asynchronous handling of interrupts in commodity OS kernels. In particular, we cannot predetermine when and where an interrupt may occur. Once an interrupt occurs, it pauses the execution of current program and forces a control transfer to the related interrupt handler. When the hardware pauses the execution of current program, it automatically saves the runtime context information such as CS and IP. Inside the interrupt handler, it might save additional context information about the interrupted program so that it can be restored later when the interrupted program is resumed. Consequently, we need to protect such context information from being misused as well. Note that since interrupts may occur at the boundary of almost any instruction, it represents an important exception to the CFG mode of the kernel code. Therefore, it is necessary to use a mechanism different from the indexed hooks to protect the information pushed to the stack by the interrupt hardware. In our implementation, we use the hypervisor to protect and verify them. The details are in Section III.

III. Implementation

To validate our approach, we have implemented a proof-of-concept prototype. In this section, we will discuss specific implementation details as well as some additional notes we observed in the process of developing the prototype for the support of multitasking FreeBSD 8.0 OS kernel.

Our prototype is developed on top of the open-source LLVM compiler framework [15]. The LLVM framework is extensible and allows us to add various compiler transformations and optimizations at different phases. In particular, our prototype involves the LLVM's back-end and takes advantage of its built-in high-quality code generator [19]. More specifically, by taking a modular design, the LLVM code generator has been divided into several stages: instruction selection, scheduling and formation, SSA-based optimization, register allocation, prologue/epilogue code insertion, late machine code optimization, and code emission. Our implementation of indexed hooks is at the code emission phase to avoid causing any conflicts with various heuristics in the late machine code optimization.

We have used our prototype to compile FreeBSD 8.0/x86-amd64 and verify its effectiveness in kernel control data protection. The compiled FreeBSD kernel runs as a guest on top of the Xen hypervisor [20] (version 3.4.1), which is assumed to be trusted, in our prototype, to protect kernel code and static kernel data. Also, it is extended to protect the jump tables, i.e., the function tables and the return tables, as well as others (such as *GDT/IDT*—Section II-D).

A. Kernel Control Data Transformation

As mentioned earlier, indexed hooks aim to limit the jump targets allowed from the control data, including all function pointers and return addresses. To achieve that, we define a new target machine class for the new instrumented *direct call*, *indirect call*, *indirect jmp*, *ret*, as well as related function pointer-accessing instructions (e.g., *mov* or *lea*). In a nutshell, the new class will traverse every instruction in the IR tree to identify and substitute the original *direct call*, *indirect call*, *indirect jmp*, and *ret* instructions. Also, it will traverse the operands of other instructions to recognize function pointers so that they can be replaced with their own indexes. In the meantime, we point out that for some function pointers which are directly initialized in global variables or kernel objects, the initialization will not involve any above instructions. In other words, our instrumentation at the code generation phase will not recognize and replace them with indexes. Instead, we observe that the initialized function pointers are contained in the symbol table. Accordingly, we examine the symbol table to identify and replace them with corresponding indexes.

We show the pseudocode to implement indexed hooks in Fig. 5. Essentially, a *direct call* is replaced with *push $ret_table_index*; *jmp dst* (lines 05 and 07) while an *indirect call* is replaced with six instructions (lines 12–14, 16–18). In particular, the instrumentation at the 13th and 14th lines (marked by †) aims to limit the range of *func_table_index* (function table index) in case of potential index overflow

```
01  for (each instruction Inst in each basic block in func f) {
02    switch (Inst) {
03      case direct call:
04        assign a ret_table_index
05        add push $ret_table_index
06        get dst from Inst
07        add jmp dst
08        delete Inst
09        break;
10      case indirect call i:
11        get dst from Inst
12        add mov (dst), %reg
13†       add cmp $func_table_size_i, %reg
14†       add jg err_handler
15        assign a ret_table_index
16        add push $ret_table_index
17        add shl $0x3, %reg
18        add jmp *FuncTable_i(%reg)
19        delete Inst
20        break;
21      case indirect jmp j:
22        get dst from Inst
23        add mov (dst), %reg
24†       add cmp $func_table_size_j, %reg
25†       add jg err_handler
26        add shl $0x3, %reg
27        add jmp *FuncTable_j(%reg)
28        delete Inst
29        break;
30      case ret:
31        add pop %reg
32†       add cmp $ret_table_size_f, %reg
33†       add jg err_handler
34        add shl $0x3, %reg
35        add jmp *RetTable_f(%reg)
36        delete Inst
37        break;
38      case others:
39        for (each operand Op of Inst)
40          if (Op is associated with a function symbol)
41            replace Op with a func_table_index
42        break;
43    }
44  }
```

Fig. 5. Pseudocode for kernel control data transformation and protection.

attacks. An *indirect jmp* is replaced with five instructions (lines 23–27). Similar to the *indirect call* case, the 24th and 25th lines (marked by †) are included to limit the range of function table index, which is defined by the function table size. Note that we do not need to instrument any direct jmp instructions because such instructions will neither invoke a function pointer nor involve a return address to return back.

Also shown in our pseudocode, a *ret* instrumented is transformed into five instructions (lines 31–35). Similar to the *indirect call*, the 32nd and 33rd lines (marked by †) are to limit the range of *ret_table_index* (return table index), which is correspondingly defined by the size of the return table. Note the above instrumentation is applied for *near ret* only, which returns to a procedure located inside the current code segment. However, there is another return called *far return* or *lret* that returns to a calling procedure in a different segment. Compared to the near *ret*, *lret* pops a target CS and IP from the stack. If the new code segment is less privileged than the current code segment, the stack pointer is incremented by the number of bytes indicated by the immediate operand, if present. In other words, a new SS and SP might also be popped from the stack.

In our running FreeBSD 8.0 kernel, there are 8329 return instructions in total. Among them, there are 8328 near *ret*s and one *lret*. The only case of *lret* occurs in the "lgdt" procedure, which is used to load the system's global descriptor table. Specifically, before returning from "lgdt," it will reload the code selector by turning the return into an intersegment return with three instructions *popq %rax*; *pushq $KCSEL*; *pushq %rax*: namely, it first pops off the return address from stack with *popq %rax*; then it pushes CS and return address through two *pushq* instructions; after that, it returns by executing the *lretq* instruction which pops both CS and IP from the stack. To instrument this *lret* instruction, we simply replace it with a long jmp instruction. Note that to prevent *lret* instruction from being misused, we have to protect both the segment selector and the instruction pointer. However, our scheme is sufficient in this case since the segment selector is fixed at KCSEL, which is the code segment selector for the FreeBSD kernel.

In order to apply hardware-based page-level protection, the jump tables are put together and page-aligned. All unoccupied entries in these tables are filled with the address of an error handling function to trap invalid indirect calls and returns. In Fig. 6, we show a real example of our instrumentation. In the example, in addition to an *indirect call* and a *ret*, there is a *lea* instruction whose first operand is a function pointer, i.e., 7607(%rip) that points to $0xffffffff8014b020$—the address of *AcpiEvGpeXruptHandler* function. This function pointer is assigned with an index—0×48 through a *mov* instruction. In our specific FreeBSD 8.0/x86-amd64 kernel, if we can count the number of related control transfer instructions that have been instrumented, we have 38 603 *direct call*, 2093 *indirect call/jmp*, and 8329 *ret* instructions. Also, among the overall 7852 function routines in the entire kernel image, 2836 of them are indirectly invoked.

B. Jump Table Construction

We have mentioned earlier that indexed hooks are orthogonal to the generation and precision of the CFG input. In our prototype, we implemented two schemes of jump tables, which reflect the kernel CFG with two different granularities. Specifically, the first scheme is coarse-grained with two large jump tables: one function table and one return table. The function table is used by all the indirect call/jmp instructions. It contains the addresses of all the functions that may be indirectly called. The return table instead is used by all the return instructions and includes all the legal return addresses. Note that this scheme can be implemented by collecting indirectly called functions and valid return addresses during compilation or linear scan of the kernel binary. Overall, we have 2836 entries in the function table and 40 696 (38 603 + 2093) entries in the return table.

In the second scheme, we construct a CFG by combining dynamic analysis and conservative CFG and then compute jump tables as described in Section II-C. More specifically, we run the same FreeBSD virtual machine under QEMU [21] to profile

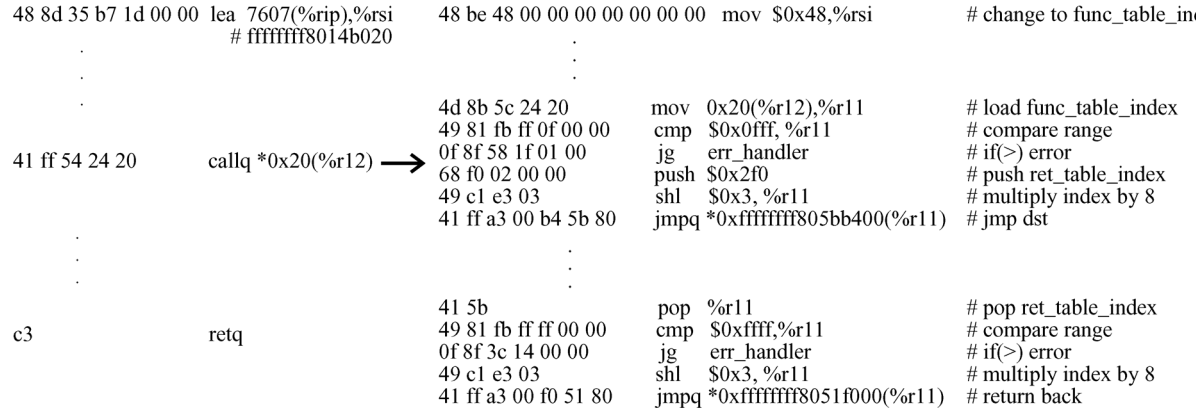


Fig. 6. Instrumentation example.

the targets of indirect calls. As dynamic analysis has an incomplete coverage, there are some indirect calls that may not be executed. For these indirect calls, we conservatively assume that they can reach all the functions that may be indirectly called. With this CFG as input, we accordingly construct both function tables and return tables. In particular, there is one function table for every indirect call/jmp instruction and one return table for each function. Note that there is no need to use more than one return table for one function because all the return instructions in a function have the same targets. Our experiment shows that 893 of 2093 (42.67%) indirect calls are reached in our profiling, and 1450 of 2836 (51.13%) indirectly called functions are called through them. The indirect call with most targets (135 targets) is in the *mi_startup()* function responsible for system initialization. Among the 893 reached indirect calls, 681 of them have only one target (76.26%). In our prototype, we have reserved 70 MB for all the function/return tables to implement the second scheme.

In comparison, the first scheme is simpler to implement as the function table and jump table can be straightforwardly derived from the kernel source or binary. However, the protection it offers is also less stricter than the second scheme. For example, in the first scheme, a return instruction can return to any of the 40 696 return sites. Therefore, it might be possible for the attacker to locate some useful gadgets out of them. On the other hand, the second scheme limits control transfer to the targets allowed by the CFG, therefore, severely limiting the attacker's possibility to locate and make use of gadgets.

C. Additional Prototyping Notes

Interrupt Handling: Like other multitasking UNIX-like kernels, FreeBSD is designed to give most interrupt handlers their own thread contexts [22]. This has the benefit in allowing them to block on locks. To help avoid latency, interrupt threads run at the real-time kernel priority. However, the interrupt threads currently in FreeBSD are considered heavyweight because switching to an interrupt thread involves a full context switch. To mitigate this, some handlers that are called "fast" ones execute directly in the primary interrupt context. These interrupt handlers include clock and serial I/O device interrupts.

As the control data involved in the normal interrupt threads will be transparently protected by our scheme, no additional processing will be needed. However, the "fast" interrupts execute in the primary interrupt context and their context information, which is automatically saved onto the stack by hardware, needs to be protected (Section II-D). In our current prototype, there are eight such interrupt handlers: *Xtimerint* for clock, and *Xapic_isr1–Xapic_isr7* for I/O devices. Rather than having one entry point for each I/O interrupt source, FreeBSD 8.0 uses one common entry point in the interrupt handler. To support nested interrupts, we use a FILO (first in, last out) queue to record and verify guest interrupt contexts in Xen. Specifically, in our current prototype, a hypercall is made at the time of an interrupt occurs. The hypervisor then saves the CS/IP pair to the FILO queue. Later, before the *iret* instruction is called to return from the interrupt handling, another hypercall is made for the hypervisor to verify the correctness of CS/IP pair by comparing them to the head of the FILO queue. The hypervisor pops the latest CS/IP pair from the queue after verification. Note that there might exist a small window of race condition from the other processors if the guest is running on an SMP machine. However, this window is small and largely unpredictable. Also, there is no need to worry about unintended modification of CS/IP on the local CPU as the interrupts are disabled before the CS/IP is pushed onto the stack.

Kernel Extension Support: One important issue of kernel extension support is related to the loading of trusted kernel modules only. Fortunately, this has been addressed by existing approaches such as SecVisor [3] and our prior work NICKLE [5]. In our implementation, we simply integrate NICKLE's module loading scheme to prevent untrusted or malicious kernel modules from being loaded. As a result, we focus our discussion on the additional challenges that indexed hooks meet to support dynamic module loading.

In particular, the main challenge comes from unifying jump tables in current kernel and loaded/unloaded LKMs. As described in Section III-B, in order to derive the jump tables, we need to obtain the kernel control-flow graph that is now affected by loading or unloading a kernel module. To provide better flexibility in supporting LKMs, the base kernel and modules may be separately compiled. To accomplish that, when a module is being compiled, we produce module-specific function tables and return tables. Instead of being populated with absolute memory addresses, the tables contain only relative offsets from the module base address. When the module is loaded, a fix-up

routine runs to reflect the current module base address in these tables. To incorporate the module's jump tables to the kernel, we first merge the two CFGs, then compute the new jump tables from the merged CFG. The indexes of kernel's control data are assigned first to keep them from being changed across the merge. In the merging process, we might need to relocate the affected jump table if its size increases beyond that allocated for it. As a result, by merging the jump tables of the module and the base kernel, we can still obtain a unified view of the jump tables.[2] On the other hand, when a module is unloaded, its index values in the base kernel are temporarily replaced with the error handling routines to capture now defunct or illegitimate control transfers. We could eliminate these unused entries from the base kernel's jump tables by removing all the external indexes and remerging existing kernel modules' jump tables. However, we never find it necessary during our experiments. Note that the fix-up routine is security-critical. Our current prototype relies on the hypervisor to prevent its execution from being tampered with. Also, the fix-up routine can only be legitimately invoked when a module is being loaded or unloaded and thus can be safely rejected at any other time.

Context Switches and User Signal Trampoline: Due to the changed stack frame convention, our scheme inevitably affects some assembly instructions that are written based on the traditional stack frame convention. There are two primary examples: one is in the CPU context switch code and another is in the user signal trampoline routine.

Specifically, in the case of a context switch, before a new process can switch in for execution, the state of the current (or old) process will be saved so that it can be restored later to continue execution. The saved state includes all the machine registers that the process may be using, such as the program counter and other OS-specific states. As a concrete example, in the FreeBSD 8.0/x86-amd64 kernel, the function responsible for performing context switch is *cpu_switch*. When it is called by the scheduler, it will store the state of the current previous process into the process control block (PCB) by a series of *movq* instructions. Among them, the program counter of the process will be saved by two instructions: *movq (%rsp)*, *%rax; movq %rax*, *PCB_RIP(%r8)*. Later on, when this process is resumed, it restores the saved state from its PCB with several similar *movq* instructions but in an opposite direction. With the changed stack frame, this becomes problematic because the saved program counter is now a return table index. In our prototype, we, therefore, add a translation instruction right before loading the program counter (IP) for the next process, which essentially converts the return table index back to the original return address form.

Similarly, for the user signal trampoline case, a signal handler is defined as a user-mode routine that the kernel will invoke when the signal is received. In particular, when a signal is being delivered, the kernel first places a signal context as well as a kernel signal-handler context frame on the user's stack. After that, it starts to run the signal trampoline code, which eventually executes an indirect call instruction (*call *SIGF_HANDLER(%rsp)*) to invoke the user's signal handler. Once it is finished, the user signal handler returns and calls the *sigreturn* system call to restore the previous signal context, hence resuming the user's process at the point where it was running before the signal occurred. As mentioned earlier, the indirect call instruction, i.e., *call *SIGF_HANDLER(%rsp)*, is involved in the above process, and by default it will be accordingly instrumented. Unfortunately, to our surprise, this causes many processes to exit. There are two reasons for this problem: First, the content stored in **SIGF_HANDLER(%rsp)* is a function address in user space but not a function pointer in the kernel. Second, this call instruction will return back from a user signal handler with a real return address, but not a return table index. In our current prototype, we choose not to instrument this particular instruction and essentially stick to the traditional stack frame convention in the user signal trampoline code. This risk is acceptable, because this jump only occurs in the user context, so it cannot be used as a kernel infection vector.

[2]In our current prototype, we support separate compilation of LKMs from the base kernel. The prototype so far allows for the coarse-grained CFG protection (Section III-B), but not the fine-grained CFG, which has an additional need to fix up the base kernel code (for every module loading).

IV. Evaluation

In this section, we perform security analysis and present the performance measurement results.

To locate all the control data and transform it into indexes, our prototype has added about 2600 lines of *C++* code to LLVM's back-end. Also, due to the stack frame changes (Section II-B), there is also a need to modify some of the FreeBSD 8.0 source code that may reference the stack frame according to the traditional convention. Fortunately, the changes are small and limited to six assembly files. We note that no *C* files have been modified (thus satisfying our second design goal—Section II). Within these assembly files, we have replaced 54 assembly instructions with 203 other instructions to reflect the changed stack frame convention. When compared to the original kernel image, our prototype image file size increased from 4 370 704 bytes to 4 850 828 bytes (∼11.0%) and the number of instructions increased from 675 763 to 764 837 (∼13.2%). Overall, our transformation of control transfer instructions to provide kernel control data protection involves ∼7.25% of the entire kernel code base.

We point out that our prototype so far focuses on the support of the FreeBSD kernel. However, as a compiler-based approach, we believe that our scheme is generic and can be applied to other commodity operating systems as well. In particular, when future releases of LLVM support other OSs (e.g., Linux), it is expected that the scheme described in this paper can be naturally applied.

A. Security Analysis

When performing the security analysis, it is important to note that our system assumes a trustworthy hypervisor, which properly establishes and enforces guest kernel code integrity. Under this assumption, to launch a kernel attack, the attackers are forced to misuse existing kernel code that is considered legitimate, such as in a typical return-oriented attack. In our evaluation, we consider any attack that is constructed by gadgets (that consist of only legitimate code) as a return-oriented

attack, regardless whether the gadgets are based on *ret* [6]–[8] or other *ret*-like instructions [23].

More specifically, to launch a return-oriented attack, an attacker needs to perform two steps. The first step is to control the stack and preload it with addresses of those return-oriented gadgets. The second step is to hijack a piece of control data (e.g., a function pointer or return address) to jump to the starting gadget. According to our current threat model (Section II), we assume that the attacker can successfully accomplish the first step. (This is due to the presence of exploitable software bugs in the OS kernels.) However, the attack will be blocked in the second step. The reason is that all the control data in the kernel have been converted to *indexes* and they are enforced in only taking allowed jump targets from either the function tables or the return tables.

In order to examine the effectiveness of our techniques, we ported Wilander's buffer overflow benchmark test-suite [24] and used it to perform several realistic attacks to hijack the control data in our system. As there are no publicly available return-oriented rootkits, what we did is try to simulate a return-oriented rootkit. We chose three pieces of control data as our hijacking targets, one in a function table, one in a dynamic kernel object, and the other one a return address in the kernel's stack. Our experimental results show that our system successfully prevented all of them.

Specifically, in the first attack to overwrite one entry in a function table, since all the jump tables are marked as read-only and protected by the Xen hypervisor, the write attempt immediately triggered a page fault exception with the error code 0×03. The error code indicates that an attempt is made to write to a read-only page. The second attack attempts to overwrite the *fork_return* function pointer in the *pcb2* kernel object with another instruction address (e.g., $0xffffffff8023c380$) in kernel space to hijack the control flow. Note that *fork_return* will be invoked by *fork_exit* routine through an indirect call to accomplish the fork procedure. This overwrite operation can be performed successfully because the memory that stores the *pcb2* object is writable. However, as *fork_return* has been instrumented to use an index, the control data overwrite was captured by the range check (Fig. 5) right before it is branched to a predefined error handler. Similarly, the third attack to overwrite the return address in the stack was blocked and trapped to the error handler. To further locate the instruction that causes the trap in the second and third attacks, our error handler will dump the machine context for later analysis. As a result, our prototype thwarts all these attacks that attempt to compromise the kernel control data, thus achieving our first design goal (Section II).

B. Performance

To evaluate the performance overhead of our system, we have performed benchmark-based measurements. These measurements include three application-level benchmarks and one micro-benchmark. They are: 1) a compilation task of the Apache server package [25], 2) a compilation task of the FreeBSD kernel, 3) a network throughput test using ApacheBench [26], and 4) a standard system micro-benchmark toolkits—LMbench [27].

TABLE I
SOFTWARE CONFIGURATIONS FOR EVALUATION

Item	Version	Configuration
Apache Compiling	2.2.13	configure & make
Kernel Compiling	8.0	make buildkernel
Apache Server	2.2.13	default configuration
ApacheBench	2.0.40-dev	ab -c 3 -n 1000000 <url>
LMbench	3-alpha1	default configuration

TABLE II
APPLICATION-LEVEL BENCHMARK RESULTS

Item	Original	New-c	New-f
Apache Build(s)	50.197	52.134 (3.86%)	51.925 (3.44%)
Kernel Build(s)	406.576	411.727 (1.27%)	410.524 (0.97%)
ApacheBench(req/s)	3534.47	3392.18 (4.03%)	3412.24 (3.46%)

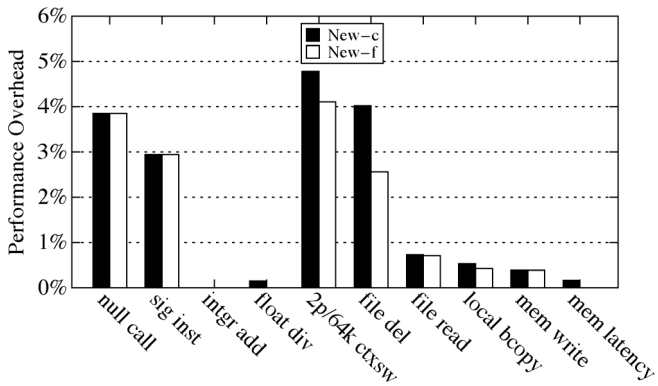


Fig. 7. Micro-benchmark results with LMbench.

Our tests were performed on a Dell Optiplex 740 PC with an AMD64 X2 5200+ CPU and 2-GB memory. For each benchmark, we load the FreeBSD 8.0 kernel with three versions: the original version (*Original*), the new kernel with coarse-grained jump tables (*New-c*), and the new kernel with fine-grained jump tables (*New-f*). Table I lists the configurations of the software used in our evaluation. In the ApacheBench test, we run an Apache server on the tested kernel and execute the ApacheBench program on a Linux client which is directly connected to our system. For each test, we ran ten times and calculated the average. The 95% confidence interval results show that the deviations among these runs are small ($<2\%$).

Table II shows the results of three application-level benchmarks. The results indicate that the overheads are less than 5%, with the maximum overhead being 4.03% in the ApacheBench test for coarse-grained kernel control data protection. Interestingly, the New-f kernel not only provides better protection, but also has *lower* performance overhead. This is possibly due to the better locality and improved cache utilization in the fine-grained jump tables. Fig. 7 shows the performance overhead of ten different kernel tasks in LMbench. The tasks include process creation, basic arithmetic operation, context switch, file system operation, local communication, and memory latency. Among these results, the maximum overheads are 4.78% and 4.10% when doing heavyweight full context switch for coarse-grained and fine-grained protection, respectively. The task of performing basic arithmetic operations incurs the lowest overhead, which is nearly zero.

To further quantify the direct impacts of instruction transformation, we apply our approach to a simple test program and measure the performance overhead. The test program simply

executes 3 million times of an indirect call to a dummy function that immediately returns. Our results show that the performance overhead from the transformation is about 33.08%. This result is expected because our approach introduces two additional memory accesses for the measured indirect call and function return (one to load the entry point of the indirectly called function and one to retrieve the return address from the jump table). This experiment represents an extreme case in terms of performance overhead.

In summary, the evaluation results show that our system introduces low performance overhead ($<5\%$ with benchmarks) based on commodity hardware support, therefore satisfying our third design goal (Section II).

V. Discussion

In this section, we discuss possible limitations of our system and suggest future refinements. First, to provide comprehensive kernel control data protection, we have taken a compiler-based approach, which requires access to the kernel source code. As a result, our approach does not support precompiled third-party drivers. Although an ideal approach would avoid such a requirement, we believe a compiler-based approach is necessary to enable the identification of all control data and then apply transformations to protect them. Without source code access, it has been shown [11], [28] that dynamic analysis-based approaches suffer from one common limitation: they cannot identify all the kernel hooks for protection. The completeness is important as rootkit authors only need to compromise *one* piece of control data to launch their attacks.

Second, it is important to note that this system does not prevent the attacker from overwriting a return table index or function table index; rather, it drastically reduces the expressiveness of this exploit, because the attacker can only choose from valid function entry points and postcall instructions allowed by the CFG. More importantly, our scheme is independent from the generation and precision of the CFG. It can take either coarse-grained or fine-grained CFG as its input. Our initial investigation shows that the wide use of *void* $*$ in commodity OS kernels complicates the CFG generation. Also, it is a challenge to scale current points-to analysis approaches (so that they are applicable for the analysis of commodity OS kernels) and enhance it for better precision (with the support of field-and-context-sensitive points-to analysis for example). Note that some promising progresses in this direction have been made by existing research efforts [18], [29], [30]. As mentioned before, our work can readily benefit from the advance in this area.

Third, it is worth mentioning that in the OS source code, we observe that a field inside a single object is not defined as a function pointer but might be later used as a function pointer. One example is the integer-based function pointer, which loads a function's address to it as an integer and later uses it as a function pointer. As pointed out in [18], this could be problematic for traditional points-to analysis approaches. Our approach does not have this problem as it automatically recognizes all function pointers. Specifically, these function pointers will be eventually used by an indirection call/jmp instruction, and our prototype will transform each and every such instruction.

Fourth, by converting control data into indexes, indexed hooks changes the semantics of function pointers and return addresses. Therefore, it might break the protected OS kernel if these data are used in the general computation, for example to serve as the base address to index into the kernel code section. Although we did not encounter such cases in our prototype, we could recover the original control data from the index before such uses.

Fifth, in this work, we assume that kernel code integrity is provided by a trustworthy hypervisor. However, we do assume the presence of exploitable vulnerabilities in the protected OS kernel. Notice that our goal here is to comprehensively protect control data of the base kernel and trusted extensions so that they always fall within the provided CFG. In the meantime, we point out that there are some promising solutions [31], [32] that are proposed to load but securely isolate untrusted kernel executions. As a result, our scheme can be naturally integrated with them for the combined benefits of ensuring the CFG of trusted kernel and isolating untrusted kernel extensions.

Finally, our goal in this paper is the protection of kernel control data, not other noncontrol data (including those branch condition data). In order to compromise noncontrol data, the attackers typically need to inject their own specific code or misuse existing kernel code. In either case, they must hijack the normal kernel control flow. As a result, our system can be naturally combined with existing systems [3], [5] for kernel code integrity to impede or prevent this attack.

VI. Related Work

Kernel Rootkit Detection and Prevention: The first area of related work covers recent efforts that aim to detect and/or prevent kernel rootkits. For example, Copilot [33] uses a trusted PCI card to grab a runtime OS image and infer if the kernel has been compromised by rootkits. The follow-up efforts extend to examine the violation of kernel data integrity [34] or state-based kernel control flow integrity [35]. Livewire [36] proposes the notion of virtual machine introspection to inspect the inner state of a guest OS to detect malware. Strider GhostBuster [37] and VMwatcher [38] leverage the self-hiding nature of rootkits to infer rootkit presence by detecting discrepancies between the views of a system from different perspectives. Other systems such as SecVisor [3] and NICKLE [5] aim to guarantee that only authenticated code can execute in the kernel space, thus providing kernel code integrity.

Closely related to our system, there are some other systems that have been proposed to protect the control data from being hijacked in either user or kernel space. For example, StackGuard [39], StackShield [40], and others [41], [42] protect the return addresses from being hijacked or misused. Lares [12] and others [11] instead protect a subset of function pointers in kernel space. In contrast, our system proposes a unified scheme that comprehensively protects all kernel control data, which includes both function pointers and return addresses. Specifically, by transforming them into their indexes, our scheme makes their protection feasible and efficient, as demonstrated in our prototype and evaluation.

Control-Flow Integrity Enforcement: There are several control-flow integrity enforcement systems proposed to protect the control data in the user space. For example, program shepherding [9] uses a dynamic machine-code translation technique (or interpreter) to constrain software's control-flow transfer. However, as pointed out in [10], the complexity of its interpreter may affect its trustworthiness and complicate its adoption. CFI [10] instead leverages a label authentication technique to guarantee that the application can jump or return to the right location for each control-flow transfer. More specifically, CFI implants certain labels at the beginning of each indirectly called function and right after each call instruction. When a control-flow transfer occurs (e.g., through an *indirect call* or *ret*), the application will compare the prestored label with the implanted label in the code to validate its correctness.

The above systems mainly target user-level applications. However, due to additional unique challenges and complexities in the OS kernel, their effectiveness in providing comprehensive kernel control data protection still remains to be shown. Also notice that CFI implants labels as *code* in the application binary, and later uses it as *data* to compare with its corresponding prestored value. Such an operation is not CPU cache-friendly and will likely introduce extra performance overhead (the highest overhead is more than 40% in the evaluation results of [10]). In comparison, our approach avoids this overhead by keeping the jump tables as static data and storing them separately from code. From another perspective, our approach essentially achieves a kernel-level control-flow integrity, which is made possible by proactively transforming kernel control data with their indexes.

Also, as mentioned earlier, our scheme is motivated from one of our prior works called HyperSafe [13], which enforces CFI for small Type-I hypervisors (e.g., BitVisor [14]). The fact that these hypervisors are designed to contain only static code and run as a single thread significantly simplifies the overall system design and implementation. In other words, for the protection of commodity OS kernel hooks, we need to accommodate the dynamic nature and address the multitasking support of commodity OS kernels, which introduce qualitative differences. As detailed in Section III, two representatives are: 1) The kernel extension support requires embedding and unifying jump tables (from the loaded kernel modules) while there is no such need in the typer-I hypervisor (as it only consists of static code); the unification process accordingly needs to intelligently handle potential conflicts in the index assignment. The reason is that it is an independent process for different kernel modules to assign their own indexes and, therefore, it is possible that same indexes can be assigned. 2) Our approach also needs to consider and gracefully handle rather frequent context switching events and interrupts, which is not the case in the single-threaded type-I hypervisor.

Other Memory Safety Protection: The third area of related work includes a number of systems to enforce memory safety for user-level applications or kernel extensions. For example, compared to CFI [10], DFI [43] further imposes the data-flow integrity to protect noncontrol data in user-level applications. XFI [44] combines static analysis with inline software guards and a two-stack execution model to perform fine-grained memory access control and it has been applied to software such as device drivers and multimedia codecs. WIT [45] combines static analysis and runtime instrumentation to prevent memory error exploits. BGI [46] uses a new software fault isolation technique to constrain the memory address space of Windows device drivers and ensure their type safety. Other systems are proposed to ensure spatial or temporal memory safety by preventing out-of-bound memory access. For example, CCured [47] and Cyclone [48] are based on fat-pointers. Each fat-pointer carries the pointer's base and bound address for run-time check. Softbound [49] goes a step further by recording base and bound information for every pointer as disjoint metadata, thus not requiring changes to source code. Others [50], [51] track allocated regions of memory as bounded objects and then map or limit pointers based on the bound information associated with the corresponding objects. SVA [52] instead proposes a new low-level, typed instruction set so that various memory safety properties can be compactly encoded as extensions to the SVA type system. To take advantage of the new typed instruction set, OS kernel and application code need to be ported to SVA. Most recently, it has been extended to account for the behavior of low-level software/hardware interactions such as memory-mapped I/O and MMU configuration [53]. Our system complements the above systems by proposing a scalable and efficient way to transform kernel control data for its protection and our evaluation indicates our approach incurs a low performance overhead.

In the same category, there also exists a long stream of researches in software fault isolation (SFI) [54]–[58]. SFI systems employ software technologies (e.g., code instrumentation, binary translation) or hardware support (e.g., segmentation in $\times 86$ architecture) to limit the data and instruction accesses of untrusted code to logically isolated parts of the host application's address space. For example, PittSFIeld [56] enforces artificial alignment rules to $\times 86$ instructions and verifies the safety of untrusted code at load time to reduce the TCB. Vx32 [57] uses dynamic instrumentation to confine instruction accesses and isolates data access with segments on the $\times 86$ architecture. SFI provides effective sandbox of untrusted code, but typically suffers from high performance overhead (especially on the $\times 86$ architecture [10]). By converting control data to indexes, our scheme could effectively protect a large number of control data with much smaller performance overhead.

VII. Conclusion

In this paper, we have presented *indexed hooks*, a scheme that comprehensively and efficiently transforms kernel control data into indexed hooks, which allows us to effectively limit them to take only the legal jump targets allowed by the CFG. Based on the observation that existing control data have a set of legitimate jump targets, we can precalculate them into (protected) jump tables and then replace these control data with their indexes to these tables. We have implemented a compiler-based approach to transform all control data in FreeBSD 8.0 kernel. Our prototyping experience and experiments with a number of synthetic attacks indicate that our scheme is generic, effective, and can be implemented on commodity hardware with a low performance overhead ($<5\%$ based on benchmarks).

REFERENCES

[1] Rootkit Numbers Rocketing UP, McAfee Says 2006 [Online]. Available: http://news.cnet.com/2100-7349_3-6061878.html

[2] Cooperation Grows in Fight Against Cybercrime 2010 [Online]. Available: http://www.avertlabs.com/research/blog/index.php/category/rootkits-and-stealth-malware/

[3] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM Symp. Operating Systems Principles*, Stevenson, WA, Oct. 2007.

[4] W^X [Online]. Available: http://en.wikipedia.org/wiki/W^X.

[5] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th Int. Symp. Recent Advances in Intrusion Detection*, Boston, MA, Sep. 2008.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proc. 15th ACM Conf. Computer and Communications Security*, Alexandria, VA, Oct. 2008.

[7] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the $\times 86$)," in *Proc. 14th ACM Conf. Computer and Communications Securityi*, Alexandria, VA, Oct. 2007.

[8] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypasssing kernel code integrity protection mechanisms," in *Proc. 18th USENIX Security Symp.*, Montreal, Canada, Aug. 2009.

[9] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proc. 11th USENIX Security Symp.*, San Francisc, CA, Aug. 2002.

[10] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Computer and Communications Security*, Alexandria, VA, Oct. 2005.

[11] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. 16th ACM Conf. Computer and Communication Security*, Chicago, IL, Oct. 2009.

[12] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. 2008 IEEE Symp. Security and Privacy*, Oakland, CA, May 2008.

[13] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proc. 2010 IEEE Symp. Security and Privacy*, Oakland, CA, May 2010.

[14] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: A thin hypervisor for enforcing I/O device security," in *Proc. 2009 ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Washington, DC, Mar. 2009.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2004 Int. Symp. Code Generation and Optimization*, San Jose, CA, Mar. 2004.

[16] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Seattle, WA, Mar. 2008.

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proc. 22nd ACM Symp. Operating Systems Principles*, Big Sky, MT, Oct. 2009.

[18] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proc. 16th ACM Conf. Computer and Communications Security*, Chicago, IL, Oct. 2009.

[19] The LLVM Target-Independent Code Generator [Online]. Available: http://llvm.org/docs/CodeGenerator.html

[20] 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 *Proc. 19th ACM Symp. Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[21] QEMU-Open Source Processor Emulator [Online]. Available: http://wiki.qemu.org/Main_Page/

[22] General FreeBSD Architecture and Design [Online]. Available: http://www.freebsd.org/doc/en/books/arch-handbook/smp-design.html

[23] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Computer and Communications Security*, Chicago, IL, Oct. 2010.

[24] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proc. 10th Network and Distributed System Security Symp.*, Feb. 2003, pp. 149–162.

[25] Apache HTTP Server Project [Online]. Available: http://httpd.apache.org/

[26] ab—Apache HTTP Server Benchmarking Tool [Online]. Available: http://httpd.apache.org/docs/2.2/programs/ab.html

[27] LMbench—Tools for Performance Analysis [Online]. Available: http://www.bitmover.com/lmbench/lmbench.html/

[28] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proc. 11th Int. Symp. Recent Advances in Intrusion Detection*, Boston, MA, Sep. 2008.

[29] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation*, San Diego, CA, Jun. 2007.

[30] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: A million lines of C code in a second," in *Proc. ACM SIGPLAN 2001 Conf. Programming Language Design and Implementation*, Snowbird, UT, Jun. 2001.

[31] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity OS from untrusted extensions," in *Proc. 18th Ann. Network & Distributed System Security Symp.*, San Diego, CA, Feb. 2011.

[32] A. Srivastava and J. Giffin, "Efficient monitoring of untrusted kernel-mode execution," in *Proc. 18th Ann. Network & Distributed System Security Symp.*, San Diego, CA, Feb. 2011.

[33] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot—A coprocessor-based kernel runtime integrity monitor," in *Proc. 13th USENIX Security Symp.*, San Diego, CA, Aug. 2004.

[34] 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 *Proc. 15th USENIX Security Symp.*, Vancouver, BC, Canada, Jul. 2006.

[35] N. L. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. 14th ACM Conf. Computer and Communications Security*, Alexandria, VA, Oct. 2007.

[36] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. 10th Annual Network and Distributed System Security Symp.*, San Diego, CA, Feb. 2003.

[37] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting stealth software with strider ghostbuster," in *Proc. 2005 Int. Conf. Dependable Systems and Networks*, Yokohama, Japan, Jun. 2005.

[38] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "Out-of-the-Box" semantic view reconstruction," in *Proc. 14th ACM Conf. Computer and Communications Security*, Alexandria, VA, Oct. 2007.

[39] 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 *Proc. 7th USENIX Security Symp.*, San Antonio, TX, Jan. 1998.

[40] Vendicator, Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux [Online]. Available: http://www.angelfire.com/sk/stackshield/info.html

[41] H. Etoh, GCC Extension for Protecting Applications From Stack-Smashing Attacks [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[42] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proc. 5th ACM SIGOPS Eur. Conf. Computer Systems*, Paris, France, Apr. 2010.

[43] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. 7th USENIX Symp. Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

[44] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. 7th USENIX Symp. Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

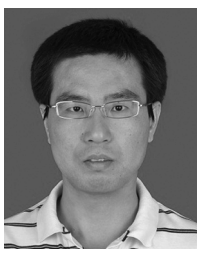
[45] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. 28th IEEE Symp. Security and Privacy*, Oakland, CA, May 2008.

[46] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proc. 22nd ACM Symp. Operating Systems Principles*, Big Sky, MT, Oct. 2009.

[47] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, pp. 477–526, 2005.

[48] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. 2002 USENIX Annual Technical Conf.*, Monterey, CA, Jun. 2002.

[49] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. ACM SIGPLAN 2009 Conf. Programming Language Design and Implementation*, Dublin, Ireland, Jun. 2009.
[50] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proc. 18th USENIX Security Symp.*, Montreal, Canada, Aug. 2009.
[51] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proc. 28th Int. Conf. Software Engineering*, Shanghai, China, May 2006.
[52] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proc. 21st ACM Symp. Operating Systems Principles*, Stevenson, WA, Oct. 2007.
[53] J. Criswell, N. Geoffray, and V. Adve, "Memory safety for low-level software/hardware interactions," in *Proc. 18th USENIX Security Symp.*, Montreal, Canada, Aug. 2009.
[54] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Operating Systems Principles*, Asheville, NC, Dec. 1993.
[55] C. Small and M. Seltzer, "MiSFIT: A tool for constructing safe extensible C++ systems," in *Proc. 3rd Conf. USENIX Conf. Object-Oriented Technologies*, Portland, OR, 1997.
[56] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proc. 15th USENIX Security Symp.*, Vancouver, BC, Canada, Jul. 2006.
[57] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the $\times 86$," in *Proc. 2008 USENIX Annual Technical Conf.*, Boston, MA, Jun 2008.
[58] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted $\times 86$ native code," in *Proc. 2009 IEEE Symp. Security and Privacy*, Oakland, CA, May 2009.



Jinku Li received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1998, 2001, and 2005, respectively.

From March 2009 to February 2011, he was a postdoctoral fellow in the Department of Computer Science at North Carolina State University, Raleigh, NC. He is currently an associate professor in the School of Computer Science and Technology at Xidian University, Xi'an, China. His research focuses on malware defense and computer system security.

Zhi Wang (S'08–M'11) received the M.S. degree in computer science from Xi'an Jiaotong University, Xi'an, China, in 2002. He is currently working toward the Ph.D. degree in the Department of Computer Science, North Carolina State University, Raleigh, NC.

His research interests include virtualization and operating system security, cloud computing security, network and web security.

Tyler Bletsch is originally from Tampa, FL. He received the Ph.D. degree in 2011 from North Carolina State University.

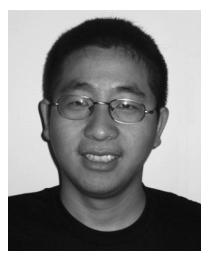
During his doctoral studies, he has worked in the fields of power aware computing, storage technology, and software security (the focus of his dissertation work). He is currently employed at NetApp as a reference architect in the Infrastructure and Cloud Engineering team.

Deepa Srinivasan received the B.S. degree in mathematics from the University of Madras, in 1997, and the M.S. degree in computer science from the Oregon Graduate Institute, in 2003. She is currently working toward the Ph.D. degree in the Department of Computer Science, North Carolina State University, Raleigh, NC.

Previously, she worked at IBM Corp. as an Advisory Software Engineer in the System x Server group.

Michael Grace (S'10) received the B.S. and M.S. degrees in computer science from North Carolina State University, Raleigh, NC, in 2006 and 2008, respectively, where he is currently working toward the Ph.D. degree.

His research focuses on reverse engineering and system security.



Xuxian Jiang (M'06) received the B.S. degree from Xi'an Jiaotong University, in 1998, and the Ph.D. degree in computer science from Purdue University, in 2006.

He is an assistant professor of Computer Science at North Carolina State University. His research interests mainly include operating systems security and malware defense. He is a member of the ACM.