Defeating Return-Oriented Rootkits With "*Return-less*" Kernels

Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, Sina Bahram
North Carolina State University
{jli17, zhi_wang, xuxian_jiang, mcgrace, sbahram}@ncsu.edu

Abstract

Targeting the operating system (OS) kernels, kernel rootkits pose a formidable threat to computer systems and their users. Recent efforts have made significant progress in blocking them from injecting malicious code into the OS kernel for execution. Unfortunately, they cannot block the emerging so-called *return-oriented rootkits* (RORs). Without the need of injecting their own malicious code, these rootkits can discover and chain together "return-oriented gadgets" (that consist of only legitimate kernel code) for rootkit computation.

In this paper, we propose a compiler-based approach to defeat these return-oriented rootkits. Our approach recognizes the hallmark of return-oriented rootkits, i.e., the *ret* instruction, and accordingly aims to completely remove them in a running OS kernel. Specifically, one key technique named *return indirection* is to replace the return address in a stack frame into a return index and disallow a ROR from using their own return addresses to locate and assemble return-oriented gadgets. Further, to prevent legitimate instructions that happen to contain return opcodes from being misused, we also propose two other techniques, register allocation and peephole optimization, to avoid introducing them in the first place. We have developed a LLVM-based prototype and used it to generate a *return-less* FreeBSD kernel. Our evaluation results indicate that the proposed approach is generic, effective, and can be implemented on commodity hardware with a low performance overhead.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Security kernels

General Terms Security

Keywords Return-Oriented Rootkits, Malware Defense, Return-less Kernel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'10, April 13–16, 2010, Paris, France.
Copyright © 2010 ACM 978-1-60558-577-2/10/04. . . $10.00

1. Introduction

Kernel rootkits pose a formidable threat to computer systems. Designed to fundamentally subvert the operating system (OS) kernels, a kernel rootkit is capable of obtaining and maintaining an unrestricted control and access within the compromised system, ranging from stealing sensitive personal information, escalating privileges of malicious processes, and opening system backdoors for unauthorized accesses. Worse, all of them can be potentially performed under the radar of running anti-virus software.

To address the rootkit threat, a variety of anti-rootkit mechanisms have been proposed in two main categories. The first category [20, 34–36] detects the presence of kernel rootkits based on certain symptoms exhibited from rootkit infection. For example, Copilot [35] leverages a separate trusted PCI card to periodically grab the physical memory image of a running OS kernel. The memory image will then be analyzed to examine whether certain properties have been violated (e.g., the checksum of static kernel text has been changed). Other systems [34, 36] further extend it by detecting the violation of kernel data integrity (e.g., by using semantic specifications of both static and dynamic kernel data) and/or kernel control-flow integrity (e.g., by comparing with a statically-computed control-flow graph from kernel source code). However, by design, these approaches detect a kernel rootkit's presence *after* the system is compromised.

The second category [18, 38, 40] instead aims to preserve the OS kernel integrity and prevent kernel rootkits from infecting the system in the first place. For example, techniques such as driver signing [1] and other forms of driver verification [23] have been proposed to verify the identity or integrity of the loaded driver. The W⊕X hardware support allows to mark a memory page as writable or executable, but not both at the same time. In other words, it will prevent kernel rootkits from injecting rootkit code as *data* into the OS kernel and later executing it as *code*. SecVisor [40] is a hypervisor-based approach that leverages W⊕X to achieve lifetime guest kernel code integrity. NICKLE [38] makes a step further by accommodating the presence of mixed kernel code and data in commodity OS kernels. More specifically, NICKLE maintains a separate, guest-inaccessible shadow memory to store authorized kernel code and at runtime, the kernel instruction fetch will be transparently redirected to

the shadow memory. Such a guarantee makes an important step in kernel rootkit prevention by effectively blocking existing kernel rootkits that require executing their own attack code.

Unfortunately, such a guarantee is still insufficient as it cannot block emerging so-called return-oriented rootkits. These rootkits are designed to re-use existing (and thus "good") kernel code for malicious (or "bad") computations *without* violating kernel code integrity. An example in the user-space counterpart is the classic *return-into-libc* attacks where the library functions (e.g., *system()*) are misused to launch or facilitate the attack. This attack has been recently refined and generalized as *return-oriented programming* [11, 19, 41]. With this programming model, the attacker can develop kernel rootkits by only misusing legitimate kernel code (i.e., with chained execution of several pieces of existing instructions or "gadgets" [11, 19, 41] – an example gadget will be presented in Section 2). Due to the departure from previous code-injection based rootkit techniques, a new term *Return-Oriented Rootkit* (ROR) [19] has been coined to represent them.

By "using" only legitimate kernel instructions, RORs pose a significant challenge for rootkit defense. Also notice that in certain hardware architecture such as *x86*, a ROR would exploit variable-length encoding and unaligned execution of machine instructions to uncover "new" instructions, such as by starting from the middle of an instruction (Figure 1(a)). These new instructions are not intended when the original kernel code is generated. In other words, if these instructions happen to end with a *ret* instruction, they can be potentially combined to form a gadget – as an organizational unit in return-oriented programming – and perform some (malicious) primitive operations, such as a comparison of two register operands or a memory load from a specified location. With a large codebase such as from a standard C library or from an OS kernel image, this new attack has shown to be Turing-complete [11, 19, 41]. In fact, with successful demonstration, a return-oriented compiler has been designed and developed to perform a variety of computational tasks by only utilizing assembled gadgets. Consequently, it becomes evident that a ROR can intrinsically bypass all existing kernel code integrity protection mechanisms, including W⊕X [3], SecVisor [40], and NICKLE [38].

In this paper, we propose a compiler-based approach to defeat return-oriented rootkits by directly attacking the root of the return-oriented programming model. In particular, based on the observation that the gadget is the essential basic unit in return-oriented programming and each gadget has to end with a *ret* instruction (so that one gadget can be chained together with other gadgets), we propose to re-target the compiler design to generate an OS kernel without *ret*. By doing so, the new OS kernel will be essentially immune to return-oriented rootkits. Naturally, our approach can be combined together with existing approaches that guarantee kernel code integrity to provide a comprehensive rootkit prevention solution.

However, it is not an easy task to eliminate these return instructions. For example, an intuitive approach to remove *ret* would involve replacing it with a *pop %eax, jmp *%eax* sequence. Unfortunately, the fact that it is semantically equivalent to a return instruction means it can still be used to build return-oriented gadgets.

To address that, we propose a key technique called *return indirection*, which is inspired by the nature of return-oriented programming: namely it requires the attacker to supply return addresses of his choice to pinpoint and execute various gadgets. In other words, during the attack, these return addresses need to be pre-populated by the attacker (and therefore *not* saved from previous call instructions). The goal of return indirection is to essentially eliminate this capability from return-oriented rootkits. Specifically, instead of following the convention of using the de-facto return address in a stack frame, return indirection replaces it with a return index. The return index will be automatically pushed onto the stack by a previous (instrumented) *call* instruction and later popped up by an (instrumented) *ret* to locate the return address. Each return index will be corresponding to a particular entry in a centralized return address table that contains all valid return addresses permitted in the OS kernel image. Since each valid return address must point to an instruction that immediately follows a call instruction, the return address table is static and can be generated offline.

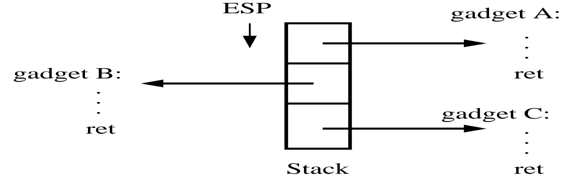
We have developed a proof-of-concept prototype that implements return indirection based on the re-targetable LLVM compiler [25]. In addition to return indirection, we also extend and refine two compiler optimization techniques, i.e., register allocation and peephole optimization, to conservatively remove machine instructions that happen to contain the return opcodes. By doing so, we can also prevent these legitimate instructions from being misused by RORs. Our system has been used to compile the FreeBSD 8.0 kernel and a few other system programs. Our subsequent examination shows there are *no* return instructions in the resulting programs and kernel images. In summary, our paper makes the following contributions:

- We perform a thorough return-oriented analysis on the FreeBSD 8.0 kernel. Our analysis covers not only the *ret* instruction, but also other instructions (e.g., *ret imm16*, *lret*, *lret imm16*, and *movnti* – Section 2) that can be potentially abused for return purposes. The analysis is needed to identify all possible sources so that they can be re-visited to *not* introduce return opcodes.
- Based on the return-oriented analysis, we propose corresponding compiling techniques and implement them in LLVM. We use the modified compiler to re-generate the FreeBSD OS kernel. The new OS kernel image is free from return opcodes (not just return instructions!) and is thus immune to return-oriented attacks.

```
AcpiUtDeleteRwLock:
  48 8b 3b          mov   (%rbx),%rdi
  e8 c3 e0 00 00    callq  <AcpiOsDeleteMutex>

  3b e8             cmp   %eax,%ebp
  c3                retq
```

(a) A return-oriented gadget starts from the middle of an instruction



(b) A return-oriented computation consists of three gadgets A,B,C

Figure 1. An example of return-oriented programming

- We evaluate the performance of our system using a number of standard benchmark programs. The results show that our prototype incurs low performance overhead.

The rest of the paper is organized as follows: We describe the return-oriented programming and RORs in more detail in Section 2. We then present the key techniques in our approach and the implementation details in Sections 3 and 4, respectively. After that, we show the evaluation results in Section 5 and discuss possible limitations of our current prototype in Section 6. Finally, we compare our system with related work in Section 7 and then conclude in Section 8.

2. Problem Overview: Return-Oriented Programming and RORs

In the return-oriented programming model, there are two central pieces: gadgets and a stack. Each gadget ends with a *ret* and performs a basic operation (e.g., a memory write to a specific location). A stack is controlled by the attacker to specify how these gadgets will be chained together.

To illustrate the return-oriented programming model, we show in Figure 1(a) an example gadget created by leveraging two existing instructions (*mov* and *callq*) in the *AcpiUtDeleteRwLock* function of FreeBSD 8.0/x86-amd64 kernel. Note the x86 architecture has a variable-length instruction set, which means the same memory content will be interpreted differently if CPU begins decoding at two distinct locations. In this example, if we start in the middle of *mov* instruction, a "new" instruction sequence is created – *cmp %eax,%ebp; retq*. This instruction sequence essentially compares the contents of two registers *ebp* and *eax* and then returns. The *retq* instruction in the new sequence is actually part of the original *callq*'s operand (the relative offset *0x0000e0c3*). It may be considered time-consuming or even infeasible to manually identify these return-oriented gadgets. However, as shown in [41], an automated process can be easily constructed to exhaustively list all candidate gadgets. Also, by examining the standard C library and commodity OS kernel images, several previous studies [11, 19, 41] report the abundant availability of various gadgets to perform basic operations (e.g., memory load/store, unary/binary arithmetic operation, and conditional jump), which makes return-oriented programming Turing-complete.

In Figure 1(b), we also show an example return-oriented computation based on three gadgets: *A, B, C*. These three gadgets are chained together to perform a specific computation. The chaining is made possible with a stack that is pre-populated with three stack frames. Each of these stack frames will be pointing to one of these gadgets. The execution order of these gadgets will be determined by the stack pointer or ESP. Specifically, the stack pointer will be advanced by the *ret* instruction of a previous gadget before other subsequent gadgets can be executed. In essence, the stack pointer becomes the new instruction pointer in the return-oriented programming model. From another perspective, the *ret* instruction is the hallmark of return-oriented programming: it needs to be present in every gadget and is also the key to chain together various gadgets (with a controlled stack).

Based on the return-oriented programming model, a successful ROR attack that compromises a running OS kernel will require to accomplish two steps. The first step is to control a stack and pre-load it with addresses of those chosen return-oriented gadgets. The second step is to hijack the kernel control flow to jump to the starting gadget. Consequently, we can defeat ROR attacks either by removing the attacker's abilities to form and chain gadgets or by preventing control flow from being hijacked in the first place. Note there are several research prototypes (e.g., [6, 22]) that can effectively enforce control-flow integrity for user-level applications. However, their application and portability to enforce the kernel control-flow integrity still remains to be demonstrated. In this work, we take the first approach. Specifically, we aim to eliminate all the *ret* instructions in a way that will make return-oriented programming infeasible. As pointed out earlier, in certain hardware architecture such as x86, because of the variable-length encoding and unaligned execution of machine instructions, it is not sufficient to only consider those return-related instructions. Instead, we also need to take into account other instructions that happen to contain the corresponding return opcodes in the generated machine code. Note that in x86, there are four return-related instructions: *ret* (near return – opcode *c3*), *ret imm16* (near return with stack unwind imm16 bytes – opcode *c2*), *lret* (far return – opcode *cb*), and *lret imm16* (far return with stack unwind imm16 bytes – opcode *ca*). For simplicity, we will

#	machine code	instruction
8328	c3	retq
0	c2 *xx xx*	retq *imm16*
1	48 cb	lretq
0	48 ca *xx xx*	lretq *imm16*

Table 1. *Return opcode source I* (Total: $8,329$): real *ret* instructions

#	machine code	instruction
1	48 0f c3 04 17	movnti %rax, (%rdi, %rdx, 1)
1	48 0f c3 44 17 08	movnti %rax, 0x8(%rdi, %rdx, 1)
1	48 0f c3 44 17 10	movnti %rax, 0x10(%rdi, %rdx, 1)
1	48 0f c3 44 17 18	movnti %rax, 0x18(%rdi, %rdx, 1)
1	48 0f c3 04 16	movnti %rax, (%rsi, %rdx, 1)
1	48 0f c3 44 16 08	movnti %rax, 0x8(%rsi, %rdx, 1)
1	48 0f c3 44 16 10	movnti %rax, 0x10(%rsi, %rdx, 1)
1	48 0f c3 44 16 18	movnti %rax, 0x18(%rsi, %rdx, 1)

Table 2. *Return opcode source II* (Total: 8): other machine instructions' opcodes that happen to contain the return opcode value

use *ret* in the rest of this paper to represent all of them and accordingly *return opcode* to represent *c2, c3, ca*, or *cb*.

We have profiled a stock FreeBSD/x86-amd64 8.0 kernel image and calculated the statistics of return opcodes. In our calculation, we do not follow instruction boundaries and instead mimic the return-oriented programming to identify all possible return opcodes. The OS kernel image has $675,763$ instructions in total and contains $18,330$ return opcodes. On average, there will be one return opcode in every 37 machine instructions.

Among those return opcodes, we further make a breakdown to understand how the return opcodes are introduced. Our results show that $8,337$ of those $18,330$ return opcodes are introduced by normal instructions' opcodes and the rest $9,993$ are due to normal instructions' operands. In those $8,337$ opcodes, $8,329$ are actually from the *ret* instruction itself (*return opcode source I*) and 8 are from other machine instructions, more specifically *movnti* (*return opcode source II*). We show those related instructions in Tables 1 and 2, respectively.

Regarding those $9,993$ return opcodes introduced by instructions' operands, there are two contributing sources: (1) $2,923$ are due to immediate operands that happen to contain the same value with return opcodes (*return opcode source III*). In Table 3, we show six instructions that contribute the largest number of return opcodes because of immediate operands. (2) The other $7,070$ are introduced by register operands in a number of machine instructions (*return opcode source IV*). For example, *c3* and *cb* will be introduced when registers *rbx* and *r11* (including their sub-registers *ebx, bx, bl*) are being used in various *mov* instructions, whereas *c2* and *ca* will be introduced due to the usages of *rdx* and *r10* (including sub-registers *edx, dx, dl*). Similarly, we list the top six instructions in Table 4.

#	machine code	instruction
52	eb cb	jmp 0xffffffff80144378
44	eb c2	jmp 0xffffffff80150164
40	eb c3	jmp 0xffffffff8014cef0
39	eb ca	jmp 0xffffffff801466b4
33	75 cb	jne 0xffffffff80146f40
25	0f 84 cb 00 00 00	je 0xffffffff8014af6f

Table 3. *Return opcode source III* (Total: $2,923$): immediate operands that happen to contain the return opcode value (Note that the disassembled *jmp/jne/je* instructions could depend on relative offsets between the location of jump target and the current instruction pointer.)

#	machine code	instruction
832	89 c3	mov %eax, %ebx
445	89 c2	mov %eax, %edx
373	48 89 c2	mov %rax, %rdx
338	48 89 c3	mov %rax, %rbx
333	48 89 cb	mov %rcx, %rbx
267	89 ca	mov %ecx, %edx

Table 4. *Return opcode source IV* (Total: $7,070$): certain register operands that happen to be encoded with the return opcode value

To disallow the creation of RORs, we need to revisit these four contributing sources so that return opcodes will not be introduced. Each source is considered to have its own unique challenges and thus likely requires different solutions. For example, for return opcode source I, it will not work by simply having *ret* replaced with other semantically-equivalent instructions. As mentioned in Section 1, the reason is that the new semantically-equivalent instruction(s) can be equivalently used as *ret* for ROR purposes. For return opcode source IV, we need to adjust the register allocation when target code is being generated so that we can avoid introducing the return opcodes. Considering the large number of instructions affected, it becomes a significant challenge to address these four contributing sources securely and efficiently.

3. System Design

3.1 Design Goals and Assumptions

In order to effectively defeat RORs, we have three main design goals. *First*, the proposed techniques aim to enable proactive prevention of the ROR attacks, instead of reactive detection of their presence after the strike. Specifically, our goal here is to create a return-less kernel without a sin-

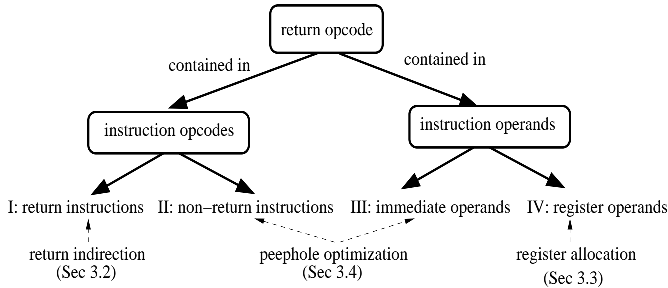


Figure 2. An overview of return opcode sources and our defense techniques

gle instance of return-oriented gadgets, thus making return-oriented programming infeasible.

Second, the proposed techniques should require minimal or ideally no modification to the OS kernel for return-oriented attack prevention. As such, the proposed techniques can be generically portable to a range of commodity systems, thus mitigating the new threats from return-oriented programming.

Third, with the advent of RORs, there is a pressing need that the proposed techniques can be efficiently implemented and readily deployable on commodity hardware, i.e., without the need of sophisticated hardware support for additional features or acceptable performance. Given this, the challenge is to ensure that our approach has a small footprint and remains lightweight with respect to performance impact.

We also point out that oftentimes, it is desirable that the proposed techniques can be extended to facilitate various flexible response mechanisms, which can be activated upon the detection of an ongoing ROR execution attempt. A flexible response, for example, is to cause only the offending attack to fail without stopping the rest of the OS. In this work, since our primary focus is on the core methodology in defeating RORs, we leave these additional desirable features as future work.

Assumptions and Threat Model: In this work, we assume the system is guaranteed with its kernel code integrity (e.g., by SecVisor [40], NICKLE [38], and other W⊕X-based schemes). Similar to the threat model used in SecVisor and NICKLE, we also assume the kernel rootkit has the highest privilege level inside the system (e.g., the root privilege in a Unix system) and full access to the system memory space (e.g., through /dev/mem in Linux). However, due to the guarantee of kernel code integrity by existing tools, the kernel rootkit is forced to re-use existing kernel code for malicious computation (e.g., hiding its presence or other malicious processes).

Based on this threat model, our system builds upon existing kernel code integrity guarantees and further blocks return-oriented rootkits. In the following subsections, for each potential source (Section 2) that may introduce return opcodes exploitable by RORs, we will describe the corresponding technique (a summarized view is shown in Figure 2) and examine how it is useful in making RORs infeasible.

3.2 Return Indirection

One primary source of the return opcode is the *ret* instruction itself. In a normal run, *ret* is typically paired up with a previous *call* instruction. Specifically, if a caller wants to invoke a callee, the caller will execute a *call* instruction that will transfer the control flow to the callee (after saving the return address on the stack). After finishing its execution, the callee will execute *ret* that will transfer the flow back to the caller (based on the previously-saved return address).

However, this is not the case in a ROR attack. In order to better illustrate the difference from the chained execution of gadgets in a ROR, we show in Figure 3(a) the control flow transfers of normal *call-ret* pairs. When compared with the chained execution of gadgets (Figure 1(b)), we observe that in the chained execution of gadgets, the return address is explicitly pre-loaded by the attacker but later consumed by *ret*; while in the normal *call-ret* pair the return address is automatically saved by *call* and consumed by *ret*. This key difference exposes the "flexibility" enjoyed by RORs in choosing arbitrary return addresses to pinpoint and chain together gadgets. To eliminate such flexibility, we propose to add one level of indirection in accessing the return address, thus the name "return indirection."

Specifically, when a *call* is made, we instrument its execution to push a return index instead of the de-facto return address onto the stack. The return index will point to an entry in a centralized return address table and the entry contains the same return address, i.e., the location of next instruction after *call*. When a *ret* is executed, instead of popping up a return address from the stack for control flow transfer, we accordingly instrument its execution to obtain the return index and use it to look up in the return address table to locate the return address, which will then be used for control flow transfer. To simplify the presentation, we call this new instrumented *call-ret* pair as *call'-ret'*. At a conceptual level, *call'* and *ret'* can be simply implemented as *push $index; jmp dst* and *pop %reg; jmp *RetAddrBase(%reg)*, respectively. To illustrate the difference from normal *call-ret* pairs, we show in Figure 3(b) the detailed control flow transfer steps when involving a *call'-ret'* pair.

As discussed earlier, the main benefit of return indirection is the removal of the attacker's capability in choosing their own return addresses for gadget identification and chaining. It also has the nice side-effect in preventing the attacker from constructing a gadget by starting from the middle of a legitimate instruction. However, the downside is the overhead in initializing and maintaining the return address table and introducing an extra memory access in the *ret'* execution when compared to *ret*. Fortunately, each return address table entry will point to the next instruction after the *call'* instruction. Given the static nature of OS kernel text, we can populate offline all entries in the return address table. Accordingly,

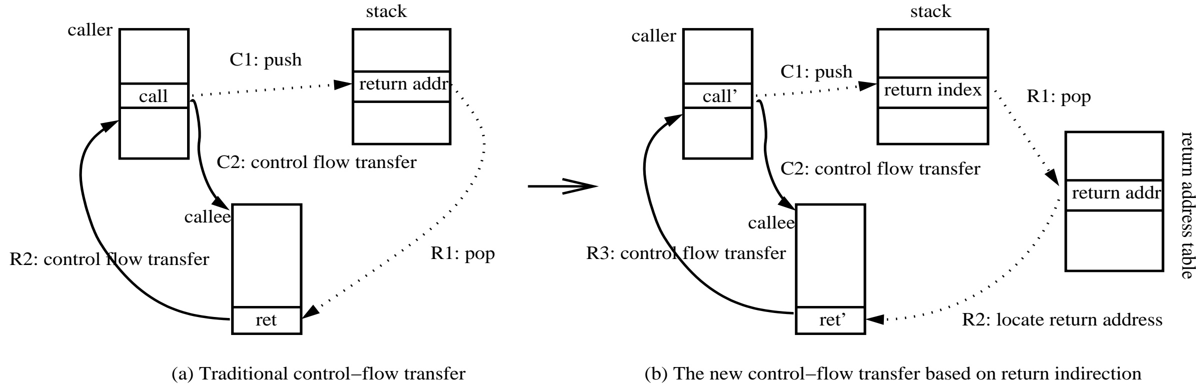


Figure 3. Traditional control flow transfer vs. the proposed return indirection-based control flow transfer

since the return address table will be static, it can be marked as *read-only* for protection (as with the static kernel text). Also, though return indirection introduces one more memory access in *ret'*, our experimental results (Section 5) show that the performance overhead is acceptable.

From another perspective, the presence of a return address table does provide another potential attack vector, which resembles the classic return-into-libc attacks. Essentially, the attacker may choose to misuse legitimate return addresses contained in the return address table. However, we note that return-oriented programming generalizes and refines the return-into-libc attacks while our return indirection technique essentially *de-generalizes* return-oriented programming back to the old style of return-into-libc. Additionally, it is known that return-into-libc attacks are restricted in executing only straight-line code, as opposed to the branching and other arbitrary behavior available with code injection. This indicates that the Turing-completeness enjoyed by return-oriented programming will be completely prevented by our technique. Furthermore, we can also add additional refinements of return indirection to further restrict this attack. For example, one possibility is to add function type-based return validation as a part of *ret'*. In essence, we can enforce that the callee will only return back to the caller if the caller invokes the function call with the same type. More details about return indirection and this particular refinement will be presented in Section 4.

3.3 Register Allocation

Another contributing source of return opcodes is from certain register usages. In particular, the way registers are allocated and operated is affected by an important compiler optimization phase called register allocation. When a program is converted from the source code into an intermediate representation (IR), the IR operates on virtual registers, which are typically not constrained by their availability. However, when the IR is eventually translated into machine code, the virtual registers used by the IR will be mapped to available physical registers. Since we only have a limited number of physical registers, this phase may inevitably lead to register spilling, i.e., saving some variables from registers to memory. To minimize the associated cost, we prefer the selection of the least frequently used variables for spilling. Unfortunately, such knowledge may not be available at compile time, which leads to the known property of register allocation as an NP-complete problem [13, 33].

To remove return opcodes introduced by certain register usages (Section 2), we are motivated to adjust the compiler's register allocation. Based on the operating granularity or effective scope, there are three main types of register allocation, i.e., local, global, and interprocedural. Local register allocation typically operates over a basic block to optimize local variable access; global register allocation considers an entire function; and interprocedural register allocation focuses on inter-procedural functional calls (e.g., this is where a calling convention is designed and enforced). The register allocation adjustment for return opcode removal can be applied in all these three levels.

Specifically, our proposed approach is to modify the register allocation algorithm to distinguish how the allocated register is being used and then adjust the allocation accordingly. For example, when a basic block includes an instruction *mov %rax, %rbx* with machine code *48 89 c3*, we need to replace the *rbx* register with another register because it is the *rbx* register that introduces the *c3* byte. When a basic block includes an instruction *mov %rbx, %rsi* with machine code *48 89 de*, there is no need to replace *rbx* as it will not introduce a return opcode. Our profiling results in Section 2 indicates that we only need to adjust two sets of registers: *rbx/r11* and *rdx/r10* in a small subset of instructions (mostly *mov*).

3.4 Peephole Optimization

The remaining two other sources are due to certain machine instructions which happen to contain the return opcode in their opcodes or immediate operands. To avoid introducing return opcodes by these instructions, we introduce a peephole optimization phase [4] when the target machine code

is being generated. Note peephole optimization is another kind of compiler optimization performed over a small set of instructions of the generated code. The name comes from the fact that the set is also called a "peephole" or a "window." This technique, though conceptually simple, has been widely used [27] since it can effectively and quickly identify and modify inefficient sequences of instructions to improve performance.

In our case, the specific peephole optimization targets the removal of return opcodes. In particular, for return opcodes introduced by non-*ret* instructions' opcodes, we simply substitute these instructions with others that bear the same semantics but do not contain any return opcode value. In fact, our analysis indicates that there is only one such instruction, i.e., *movnti mem32/64, reg32/64*, whose opcode is *0f c3* [7]. Accordingly, we can simply replace it with a regular *mov* instruction with the same semantic meaning but without return opcode in the generated machine code.

To remove return opcodes introduced by immediate operands, we further differentiate two scenarios: (1) In the first scenario, the return opcode is introduced by the direct use of immediate constants. For that, peephole optimization can alternatively use multiple steps to generate the same constant values. Using the instruction "*cmp $0xc3,%ecx*" as an example, though $0xc3$ is being used as an immediate operand in this instruction, we can achieve the same effect by the following instruction sequence, assuming %reg is available.

cmp $0xc3, %ecx:
    mov $0xc4, %reg
    dec %reg
    cmp %reg, %ecx

(2) The second scenario is due to the use of relative offsets as immediate operands. A typical example is the set of *jmp/jne/je* instructions. Similarly, consider the following instruction from a running FreeBSD 8.0 kernel image:

0xffffffff801a4f01: e9 c3 00 00 00   jmpq 0xffffffff801a4fc9

It is interesting to note that in the disassembled form, the immediate operand of the instruction is *0xffffffff801a4fc9*, which does not contain any return opcode but the generated machine code does have the return opcode, i.e., *0xc3*. It turns out that the *0xc3* is a relative offset starting from the next instruction right after *jmpq*. And *0xffffffff801a4fc9* is the jump target address automatically calculated by the disassembler. For (near) *jmp* instructions, the following instruction (rIP), the relative offset, and the final jump target address will satisfy the following equation:

destination address	=	*%rIP*	+	*relative offset*
(0xffffffff801a4fc9)		(0xffffffff801a4f06)		(0xc3)

With that, peephole optimization can simply adjust the relative offset by adding a NOP instruction right after *jmpq*. Accordingly, the corresponding instruction will be translated into:

0xffffffff801a4f01: e9 c4 00 00 00   jmpq 0xffffffff801a4fca
0xffffffff801a4f06: 90   nop

Note if the relative offset has a *c3* in a higher-order byte position, we employ two different strategies. When the relative offset has a *c3* in the second byte position, to remove the c3, we can insert up to 256 NOPs. However, padding becomes less reasonable when facing a relative jump of greater size. For example, relative offset *0x00c30000* would require $64K$ of NOPs as padding. For that, we change the link script to relocate the target function and avoid this problem. In our prototype with the FreeBSD kernel, we have not encountered this situation.

4. Implementation

The previous sections have described the overall system design and three compiler-based key techniques to prevent return-oriented rootkits. In this section, we discuss specific implementation details and some additional notes we observed when developing our prototype.

Our prototype is based on the LLVM compiler framework [25], which is designed to support transparent, lifelong program analysis and transformation for arbitrary programs. The LLVM framework is extensible and allows us to add various compiler transformations at different phrases, such as compiler-, link-, and run-time. Also, for each hardware target such as x86-32 or x86-64 (amd64), LLVM provides an abstraction layer that defines a target machine class and generates machine code based on the abstraction.

In our prototype, the three proposed techniques are all implemented in the LLVM's back-end. By doing so, we can take advantage of the built-in high-quality code generator [2]. More specifically, by taking a modular design, the code generator has been divided into several different stages: instruction selection, scheduling and formation, SSA-based optimization, register allocation, prologue/epilogue code insertion, late machine code optimization, and code emission. Our first key technique – return indirection – is implemented at the end of the prologue/epilogue code insertion phrase; the second technique – register allocation – is naturally merged into the current register allocation phase; while the third technique – peephole optimization – mainly becomes a part of the late machine code optimization phrase. In the following, we describe in detail these three techniques.

4.1 Return Indirection and Type-based Validation

To enable return indirection, we define a new target machine class that essentially replaces the normal *call-ret* pair with the new *call'-ret'* pair. In a nutshell, the replacement is achieved by traversing every instruction in the IR tree to identify and substitute the original *call* and *ret* instructions. Recall that when replacing *call*, the *call'* instruction will push the corresponding return index onto the stack. We skip those return indexes if they contain return opcodes in their values.

```
01 for (each instruction Inst in every basic block) {
02     switch (Inst) {
03         case call:
04             get target from Inst
05             assign a unique return_index
06             add push $return_index
07             add jmp target
08             delete Inst
09             break;
10         case ret:
11             add pop %reg
12             add and $0xfffff, %reg
13†            add cmpw $TypeID, TypeAddrBase(%reg)
14†            add jne err_handler
15             add jmp *RetAddrBase(%reg)
16             delete Inst
17             break;
18     }
19 }
```

Figure 4. The pseudo code for *return indirection* (†: Note lines 13 and 14 are optional and they implement the feature of using function types to validate returns.)

In Figure 4, we show the pseudo code of replacing *call-ret* with *call'-ret'* to implement return indirection. Essentially, *call* is replaced with *push return_index; jmp* while *ret* is replaced with five instructions: the *and* instruction at the $12th$ line is to limit the range of *return_index* in case of potential index overflow attacks by RORs (Section 6); the instructions at the $13th$ and $14th$ lines (marked by †) implement an optional refinement for function type-based return validation. Since the function type information can be readily obtained from LLVM IR, for each *call*, when we assign its unique *return_index*, we also obtain the next instruction (as its corresponding return address) and the callee's function type. They will be accordingly saved in the return address table *RetAddrBase* and the function type table *TypeAddrBase*, both indexed with *return_index*. With that, we can ensure the callee will only return back to the caller that invokes the callee with the same function type. In our current prototype, we reserve 1MB memory for the two tables. The two tables are physically interleaved to better exploit the temporal and spatial localities when being accessed, which can be beneficial in reducing the system performance overhead. And unoccupied entries in the tables are filled in with the address of an error handling function to trap invalid returns. In Figure 5, we show a real-world example with the effects before and after applying return indirection. Overall, we have instrumented $40,696$ *call* instructions and $8,329$ *ret* instructions for the FreeBSD 8.0 OS kernel image.

4.2 Register Allocation

In LLVM, every machine architecture has a target register description file, which holds all the available registers for allocation. Based on the register description file, LLVM implements three different register allocation algorithms: *simple*, *local*, and *linear scan*. Note both simple and local register allocation algorithms use a direct mapping from virtual registers to physical registers while the linear scan register allocation [37] performs more advanced register scanning and allocation and uses a spiller if necessary in order to place memory loads and stores, which makes it more efficient than the other two.

In our prototype, we use the linear scan register allocation as the base algorithm and add additional support to remove return opcodes due to "unsafe" register usages that may introduce return opcodes. More specifically, our prototype extends the register description file for the *x86* architecture, *X86RegisterInfo.td*, and annotates those registers that may introduce a return opcode as unsafe. When performing linear scan register allocation, if an unsafe register leads to return opcodes when used in the related machine instructions, we re-run the register allocation algorithm. In this second run, the unsafe register will not be available for register allocation.

group	#	examples
jmp	777	jmp, jmpq, je, jne, ja, jbe, jg, ...
call	1037	callq
mov	808	mov, movb, movw, movzbl, lea, ...
cmp	203	cmp, cmpq, cmpl, cmpb, cmpxchg
others	98	testb, test, and, incl, incq, decl, ...

Table 5. Groups of instructions that introduce return opcodes with immediate operands

4.3 Peephole Optimization

Peephole optimization is introduced to adjust the generated code so that no return opcodes will be accidentally introduced. As mentioned earlier, it mainly addresses those instructions whose opcodes or immediate operands happen to contain return opcodes.

In the FreeBSD 8.0 kernel, we observe only one instruction whose opcode has the return opcode: *movnti* (opcode *0f c3*). Note that *movnti* stores a value in a 32-bit or 64-bit general-purpose register into a memory location and further indicates to the processor that the data is non-temporal and is unlikely to be used again soon. Thus the processor may avoid polluting the cache by not writing the data into the cache hierarchy. For that, our peephole optimization simply replaces it with a regular *mov* instruction, whose opcode is *89* and so does not contain a return opcode. The only difference between the two is that *mov* doesn't indicate to the processor to reduce cache pollution. As *movnti* is very rarely used in the system (only 8 of them), the side-effect due to the replacement can be neglected.

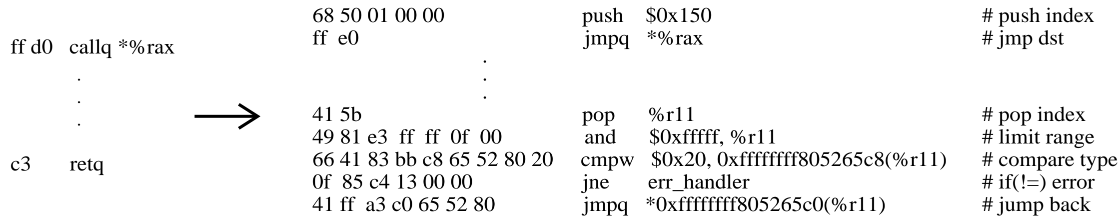


Figure 5. An example of replacing a *call-ret* pair for return indirection

For those instructions that happen to contain the return opcode value in their immediate operands, we further classify them into several groups, as shown in Table 5.

- The instructions in the first group all use relative offsets to transfer control flow to jump targets. Peephole optimization simply adds *NOP* instructions before (when jumping backward) or after (when jumping forward) the related instruction to change the relative offset value.
- The instructions in the second group are essentially *call* instructions that will be replaced with *call'* or essentially *jmp*. Therefore, it becomes the same as the first group.
- For the remaining three groups, almost all the return opcodes are introduced by direct immediate operands. The peephole optimization handles them one by one with a similar method used on *cmp $0xc3, %ecx*, which is already described in Section 3.4.

An exception is that when the immediate operand acts as an offset of the *rIP* register. For instance, *lea 0x2612c3(%rip), %rdi* will introduce a $c3$ return opcode by $0x2612c3$, which is actually a relative offset between the parameter address (the parameter will be stored in %rdi) and *rIP* register. Similar to the first group, we simply add a *NOP* instruction after *lea* to change the constant value to $0x2612c4$, thus removing $c3$.

4.4 Additional Prototyping Notes

In the following, we report some additional implementation notes we observed while developing our prototype.

CPU Context Switch Our return indirection technique essentially changes the traditional stack frame as the return address is now replaced with a return index. Such change inevitably affects other system routines that may implicitly assume the presence of return address in the stack frame. One example is the context switch routine inside the OS kernel.

Specifically, when a context switch occurs, the state of the previous process must be saved so that when the scheduler resumes its execution, it can restore the saved state without disruption. The state of the process includes all the registers that the process may be using, such as the program counter (rIP), and any other operating system specific data that may be necessary. Using the FreeBSD 8.0 kernel as an example, the function responsible for performing context switch is *cpu_switch*. When invoked, it stores the context of the previous process into the process control block (PCB) by a series of *movq* instructions. Among these instructions, it stores the return address of the process with two instructions: *movq (%rsp), %rax*; *movq %rax, PCB_RIP(%r8)*.

With the changed stack frame, it becomes problematic (because the return address is now a return index). In our prototype, we therefore add a translation instruction right before loading the rIP for the next process, which essentially converts the return index back to the original return address form.

User Signal Trampoline Like other Unix variants, FreeBSD defines a set of *signals* for certain software and hardware conditions that may arise during the normal execution of a program [28]. An application can register its own signal handlers that will be invoked when certain signals are delivered. If not, the default actions may be carried out by the system.

Notice that a signal handler is a user-mode routine that the OS kernel will invoke when a running process receives a signal. In particular, when the signal is received, the kernel first prepares a signal-handling context stack frame on the process' user stack and executes *call *SIGF_HANDLER(%rsp)* to invoke the signal trampoline code defined by the *sigtramp()* routine in the kernel, which calls the user's signal handler. Once the signal handler completes, it returns back to *sigtramp()* and pops the signal-handler context from the user's stack. After that, it calls the *sigreturn* system call to restore the previous user context and resume the execution.

We point out that return indirection by default will replace the *call *SIGF_HANDLER(%rsp)* instruction with the corresponding *call'* instruction, which unfortunately will cause many processes to exit. The reason is that return index here should be converted back to a return address as well. In our current prototype, since this happens when the signal handler returns from the user space to the kernel space, we choose not to replace this *call* with *call'*.

Compiler Optimization for Unreachable Code Elimination In the initial prototype when our system was not functioning, we also encountered several situations where certain machine instructions used to replace *call* had been "mysteriously" removed by the compiler. Consider these two consecutive instructions: *call; jmp* , which will be replaced

Item	Version	Configuration
Apache Compiling	2.2.13	configure & make
Kernel Compiling	7.2	make buildkernel
Apache Server	2.2.13	default configuration
ApacheBench	2.0.40-dev	-c 3 -n 1000000 <url>
LMbench	3-alpha1	default configuration

Table 6. Software configuration for evaluation

with *push; jmp; jmp*. Note the *push* instruction is used to save the return index onto the stack and the first *jmp* will jump to the target destination. However, the compiler will consider the second *jmp* as unreachable, thus removing it as a part of optimization. Our prototype detects that and prevents the code from being "optimized" out.

Loadable Kernel Module Support Loadable Kernel Module (LKM) support is a very important function in modern operating systems. Without LKMs, an operating system would have to have all possible anticipated functionality compiled directly into the base kernel; every time new functionality is desired, users would have to rebuild and reboot the base kernel.

In FreeBSD, LKMs can be dynamically loaded into the running kernel with *kldload*, and unloaded from the running kernel with *kldunload*. Our system supports LKMs in much the same way that it supports the base kernel. When a LKM is compiled, a module-specific return address table will be initialized. However, instead of being populated with the absolute memory addresses, the module-specific return address table contains only the offset from the module base address. At runtime, when the module is being loaded, the module-specific return address table will be fixed up based on where the module is loaded. Further, the return index assignment inside the module will also be adjusted (by a simple shift operation) to avoid causing any conflict with existing assignments in the base kernel. After that, the module-specific return address table will be added to the centralized return address table. By doing so, we have a consistent scheme for return index assignment without the need of differentiating between base kernel and loaded modules.

5. Evaluation

To generate return-less kernels, our prototype has added approximately $2,100$ lines of *C++* code to LLVM's back-end. To successfully load and run the FreeBSD 8.0 kernel, there is a need to modify the FreeBSD source code. Fortunately, the changes are minor: all the modifications focus on the assembly files (more precisely, 5 of them) and there is no single C source file that needs to be modified. Within these assembly files, we replaced 53 assembly instructions (with 232 other instructions). These changes are necessary since return indirection replaces each return address in the stack with a return index.

We also performed a return-oriented analysis on the generated kernel image. When compared to the original kernel image, the image file size is increased from $4,370,704$ bytes to $4,782,952$ bytes ($\sim 9.4\%$) and the number of instructions is increased from $675,763$ to $749,227$ ($\sim\ 10.9\%$). Most importantly, *all previous return opcodes (in total* $18,330$*) have been removed in the new OS kernel image!* The absence of return opcodes indicates that RORs, or the more general return-oriented programming model, are *prevented* from locating and building the return-oriented gadgets in the first place, which meets our first design goal (Section 2).

We point out that although our discussion so far focuses on the support of the FreeBSD kernel, we have reason to believe the technique presented here is generic and can be applied to other commodity OSes as well. In fact, we have successfully used our prototype to compile and run Xinu (an instructional OS [5]) and BitVisor (a research hypervisor [42]) with similar minor modifications in their assembly files. When the future releases of LLVM successfully compile other commodity OSes (e.g., OpenBSD and Linux), it is expected that our scheme and implementation presented in this paper is generic and can be naturally applied, thus satisfying our second design goal.

5.1 Performance

To evaluate the impact on system performance, we have performed benchmark-based measurements. In particular, we use three application-level benchmarks and one micro-benchmark to evaluate the system. They are (1) Task I: a normal compilation of the Apache server package [9], (2) Task II: a normal compilation of the FreeBSD kernel, (3) Task III: a network throughput test on the Apache web server using the ApacheBench [10], and (4) Task IV: a standard system benchmark toolkit called LMbench [26].

Our tests were performed on a Dell Optiplex 740 PC which has an AMD64 X2 5200+ CPU and 2GB memory. For each benchmark experiment, we load the FreeBSD 8.0 system with three different kernels: the original FreeBSD kernel and the new return-less FreeBSD kernel *with* and *without* the type-based return validation (Section 4.1). In Table 6, we show the configurations of the software used in our measurements. In the network throughput test on Apache, we executed ApacheBench program on a Linux client, which is connected to our system with a gigabit Ethernet card. We did all the tests ten times, then calculated the average. The 95% confidence interval results show that the deviations among these runs are small ($< 2\%$).

Figure 6 shows the measurement results of two compilation tasks (Tasks I and II) and ApacheBench (Task III), which are three application-level benchmarks. The results indicate that the overall overheads of two compiling tasks – shown by the first and third columns from left in the figure – are less than 3.5%. The overall overhead of ApacheBench is 5.78% when the type-based return validation is enabled.

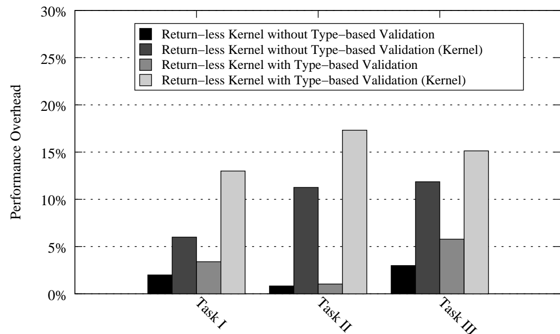


Figure 6. Application-level benchmark results

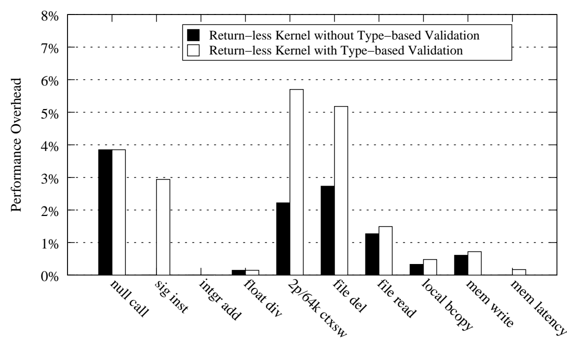


Figure 7. Micro-benchmark results with LMbench

To further measure the kernel-side impact of our system, we calculated (with the *time* command) the actual kernel-side overheads of these three tasks. The results are shown in the same figure by the second and fourth columns from left. For Task I, the kernel-side slowdown is 6.00% when we run the return-less kernel without type-based validation. If we enable the type-based validation, the overhead becomes 13.00%. For Task II, when the return-less kernel does not have the type-based validation, the kernel-side slowdown is 11.26%. With the type-based validation turned on, it changes to 17.32%. Finally, for Task III, the kernel slowdown becomes 11.86% and 15.13% accordingly when under the return-less kernel without and with type-based validation.

Figure 7 shows the performance overhead of ten different tasks in the LMbench, which is a micro-benchmark for OS kernel performance. The tasks include process creation, basic arithmetic operation, context switching, file system operation, local communication, and memory latency. Among these results, we can see the maximum overhead of our system is 5.70% when doing context switching. The overhead comes from the return indirection operation as well as an extra instruction in converting the return index back to the return address. The task of performing basic arithmetic operations incurs the lowest overhead, which is nearly zero.

In summary, the benchmark results show that our system incurs low performance overhead, which satisfies our third design goal (Section 2).

6. Discussion

To defeat return-oriented rootkits, we have taken a compiler-based approach, which naturally requires access to the kernel source code. Although an ideal approach would avoid such a requirement, we consider a compiler-based approach appropriate given the large number of return opcodes in the final OS kernel image and the resulting Turing completeness with all return-oriented gadgets. In the following, we examine possible limitations of our system and suggest future improvements.

First, as mentioned earlier, return indirection attacks the root of return-oriented programming by removing potentially-abusable return opcodes and thus de-generalizes it back to the old style of return-into-libc attacks. In the meantime, we point out our technique itself does not address return-into-libc, even though this attack is limited in executing only straight-line code and can be further limited by the proposed type-based return validation mechanism. In the meantime, another proactive approach is to call for a complete kernel control-flow integrity (K-CFI) enforcement, which should protect not only return addresses, but also various function pointers. Encouragingly, there are some existing solutions [6] that have been proposed in the user space. However, with additional unique challenges and complexities in the kernel space (e.g., memory safety regarding low-level software-hardware interactions[15], preemptive scheduling for multi-tasking, asynchronous interrupt handling, and the dynamic kernel module support), their application for complete K-CFI still remains to be demonstrated.

Second, closely related to return-oriented rootkits, one may wonder about the possibility of jump-oriented rootkits (JORs). Instead of leveraging *ret*, a JOR rootkit could leverage *jmp* to chain the execution of available machine code snippets (or "jump-oriented gadgets"). Considering the fact that *jmp* itself will not allow the attacker to regain control (since there is no *ret*), it is hard, if not impossible, to develop such rootkits. From another perspective, this attack resembles the return-into-libc attacks. However, without the leverage of *ret*, the attacker's capability will be significantly restricted.

Third, in our current prototype, we do not address another return-related instruction, i.e., *iret*. Compared to *ret*, *iret* will restore all eflags, CS, and rIP from stack. However, the fact that there are only 3 *iret* instructions in our OS kernel indicates the attacker may still need to use other return-based gadgets to build meaningful attacks.

Fourth, in our current prototype, we detect possible return index overflow attempts with an *and* $\$0xffffff$, *%reg* instruction (Figure 4), which essentially limits the range of *return_index* values. As a future refinement, we can choose

to leverage the processor's segmentation-based protection mechanism. Specifically, we can fit the entire return address table in a separate segment.[1] By properly initializing its base and size, we can take advantage of hardware-based boundary checking for improved performance and save one instruction when performing the *ret'*.

7. Related Work

Return Address Protection and Transformation The first area of related work includes a number of existing software defense mechanisms that are proposed to protect return addresses on the stack and prevent them from being misused. For example, StackGuard [14] detects attacks that overwrite the return address on the stack by inserting a marker (called a canary) between the frame pointer and the return address. Before a function returns, the canary is checked for modification. A similar scheme is also implemented in ProPolice [17]. StackShield [43] is another scheme that chooses to maintain two separate stacks for return addresses with the goal of making it significantly hard for the attacker to simultaneously modify both return addresses without being detected. SmashGuard [31] implements a similar idea but takes the approach of revising the microcode of both *call* and *ret* instructions.

These schemes are certainly helpful in protecting return addresses in existing stacks but still not sufficient. The reason is that a return-oriented rootkit, as demonstrated in [11, 19, 41], can simply create its own stack – without corrupting existing ones – and misuse other innocent non-*ret* instructions that are not (or cannot be) enhanced for return address protection. Our transformation of *ret* instructions is also similar to that of Yang et al. [46], which mainly targets to save RAM by eliminating the call stack. However, our system has a different goal that further demands two other compiling techniques to completely remove return opcodes, not merely return instructions.

Kernel Rootkit Prevention The second area of related work covers recent systems that directly aim at preventing kernel rootkit infection. Livewire [18] proposes the notion of virtual machine introspection and applies it to protect the guest OS kernel code and critical data structures from being modified. SecVisor [40] and NICKLE [38] both aim to guarantee that only verified kernel code will be running in the kernel space. Lares [32] makes a step further in protecting not only the kernel code, but also a subset of function pointers that are used for reliable active monitoring. HookSafe [45] further extends it to enable a scalable but lightweight kernel hook protection.

However, as pointed out earlier, these approaches are still vulnerable to return-oriented attacks where malicious computations are performed by leveraging existing return-oriented, "good" kernel code snippets as gadgets. Our approach complements existing ones by replacing potentially abusable return instructions and making them infeasible for malicious gadget construction and assembling.

[1] We notice that the segmentation-based support may not be available in 64-bit architectures. However, it remains enabled in its compatibility mode with 32-bit.

Kernel Rootkit Detection and Profiling The third area of related work includes recent efforts [20, 24, 34–36, 44, 48] that aim to detect and/or analyze kernel rootkit behavior. In particular, the detection can be achieved by monitoring certain symptoms from rootkit infection. For example, Petroni et al. [35] uses an external hardware PCI card to grab the runtime OS memory image and detect possible rootkit presence by spotting certain kernel code integrity violations. Strider GhostBuster [44] and VMwatcher [20] target the self-hiding nature of rootkits and infer rootkit presence by detecting discrepancies between the views of the same system from different perspectives. Other approaches such as K-Tracer [24], PoKeR [39], HookFinder [48], and Panorama [47] are proposed to analyze kernel malware so that we can extract their behavioral profiles and better understand rootkit-inherent characteristics. Our approach has a different goal by focusing on preventing a new breed of rootkits, instead of detecting or analyzing a kernel rootkit infection. Therefore, our approach is complementary to the above systems and interesting opportunities still remain to seamlessly integrate them.

Compiler-based Memory Safety The fourth area of related work includes a number of compiler-based systems to enforce memory safety for user-level applications. By instrumenting the control flow transfer instructions, CFI [6] demands the run-time control flow to follow the statically determined CFG (control flow graph). On the other hand, DFI [12] imposes the data flow integrity to defend against non-control-data attacks. CCured [30] and Cyclone [21] propose memory safe dialects of C to prevent memory corruption. Both systems require non-trivial efforts to port the existing C source code to these dialects. Another related work is software-based bound checking that intends to provide spatial or temporal memory safety by preventing out-of-bound memory access. Among bound checking systems, many of them [8, 16] are object-based. They track all allocated memory regions to map and limit pointers to individual memory region at runtime. Others (e.g., [21, 29, 30]) are based on fat-pointer. In addition to the actual pointer, each fat-pointer carries with it the pointer's base and bound.

As mentioned earlier, these systems can provide effective protection on memory safety and prevent the normal control-flow from being hijacked. However, it still remains to show how they can be applied to OS kernel protection due to additional complexities and challenges in the kernel space. From another perspective, our approach complements them by eliminating possible return-oriented gadgets in the OS kernel images and thus making RORs infeasible.

8. Conclusion

In this paper, we have presented the design, implementation and evaluation of a compiler-based approach to defeat return-oriented rootkits (RORs). Our approach recognizes the necessity of return instructions in these return-oriented attacks and accordingly proposes three key techniques (i.e., return indirection, register allocation, and peephole optimization) to completely remove them, thus making return-oriented attacks infeasible. We have developed a prototype system to generate a return-less FreeBSD kernel and the evaluation results show that our approach is generic, effective, and can be implemented on commodity hardware with a low performance overhead.

Acknowledgments We would like to deeply thank our shepherd, John Regehr, and the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. We would also like to thank Deepa Srinivasan, Minh Q. Tran, Emre Can Sezer, and Ahmed Moneeb Azab for the helpful discussion. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO and the NSF.

References

[1] Driver Signing Requirements for Windows. *http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx*.

[2] The LLVM Target-Independent Code Generator. *http://llvm.org/docs/CodeGenerator.html*.

[3] W^X. *http://en.wikipedia.org/wiki/W^X*, .

[4] Peephole Optimization. *http://en.wikipedia.org/wiki/Peephole_optimization*, .

[5] Xinu. *http://en.wikipedia.org/wiki/Xinu*.

[6] M. Abadi, M. Budiu, Úlfar Erilingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, October 2005.

[7] *AMD64 Architecture Programmers Manual Volume 3: General-Purpose and System Instructions*. Advanced Micro Devices, 3.14 edition, September 2007.

[8] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[9] Apache. Apache HTTP Server Project. http://httpd.apache.org/.

[10] ApacheBench. http://httpd.apache.org/docs/2.2/programs/ab.html.

[11] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008.

[12] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.

[13] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6:47–57, 1981.

[14] 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 *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[15] J. Criswell, N. Geoffray, and V. Adve. Memory Safety for Low-Level Software/Hardware Interactions. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[16] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering*, May 2006.

[17] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. *http://www.trl.ibm.com/projects/security/ssp/*.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network & Distributed System Security Symposium*, February 2003.

[19] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypasssing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

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

[22] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[23] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, December 2004.

[24] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, February 2009.

[25] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Pro-*

ceedings of the 2004 International Symposium on Code Generation and Optimization, March 2004.

[26] LMbench. LMbench - Tools for Performance Analysis. `http://www.bitmover.com/lmbench/lmbench.html/`.

[27] W. M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8:443–444, 1965.

[28] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2004. ISBN 0-201-70245-2.

[29] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[30] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27:477–526, 2005.

[31] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, 55:1271–1285, 2006.

[32] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[33] F. M. Q. Pereira and J. Palsberg. Register Allocation After Classical SSA Elimination is NP-Complete. In *Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structures*, March 2006.

[34] N. L. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[35] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium*, August 2004.

[36] 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 15th USENIX Security Symposium*, July 2006.

[37] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, 1999.

[38] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, September 2008.

[39] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th ACM European Conference on Computer Systems*, March 2009.

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007.

[41] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[42] 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 *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2009.

[43] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. *http://www.angelfire.com-/sk/stackshield/info.html*.

[44] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, June 2005.

[45] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communication Security*, October 2009.

[46] X. Yang, N. Cooprider, and J. Regehr. Eliminating the Call Stack to Save RAM. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2009.

[47] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[48] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, February 2008.