Detecting Code Reuse Attacks with a Model of Conformant Program Execution

Emily R. Jacobson, Andrew R. Bernat,
William R. Williams, and Barton P. Miller

Computer Sciences Department, University of Wisconsin
{jacobson,bernat,bill,bart}@cs.wisc.edu

Abstract. Code reuse attacks circumvent traditional program protection mechanisms such as $W \oplus X$ by constructing exploits from code already present within a process. Existing techniques to defend against these attacks provide ad hoc solutions or lack in features necessary to provide comprehensive and adoptable solutions. We present a systematic approach based on first principles for the efficient, robust detection of these attacks; our work enforces expected program behavior instead of defending against anticipated attacks. We define *conformant program execution* ($\mathcal{CPE}$) as a set of requirements on program states. We demonstrate that code reuse attacks violate these requirements and thus can be detected; further, new exploit variations will not circumvent $\mathcal{CPE}$. To provide an efficient and adoptable solution, we also define *observed conformant program execution*, which validates program state at system call invocations; we demonstrate that this relaxed model is sufficient to detect code reuse attacks. We implemented our algorithm in a tool, ROPStop, which operates on unmodified binaries, including running programs. In our testing, ROPStop accurately detected real exploits while imposing low overhead on a set of modern applications: 5.3% on SPEC CPU2006 and 6.3% on an Apache HTTP Server.

Keywords: Binary analysis, static analysis, return-oriented programming, jump-oriented programming.

1 Introduction

Code reuse attacks are an increasingly popular technique for circumventing traditional program protection mechanisms such as $W \oplus X$ (e.g., Data Execution Prevention (DEP)), and the security community has proposed a wide range of approaches to protect against these attacks. However, many of these approaches provide ad hoc solutions, relying on observed attack characteristics that are not intrinsic to the class of attacks. In the continuing arms race against code reuse attacks, we must construct defenses using a more systematic approach: good engineering practices must combine with the best security techniques.

Any such approach must be engineered to cover the complete spectrum of attack surfaces. While more general defensive techniques, such as Control Flow

J. Jürjens, F. Piessens, and N. Bielova (Eds.): ESSoS 2014, LNCS 8364, pp. 1–18, 2014.

Integrity or host-based intrusion detection, provide good technical solutions, each is lacking in one or more features necessary to provide a comprehensive and adoptable solution [51]. We must develop defenses that can be effectively applied to real programs.

We present a technique based on first principles for the efficient, robust detection of code reuse attacks. Our work is grounded in a model of conformant program execution ($\mathcal{CPE}$), in which we define what program states are possible during normal execution. We generate our model automatically from the program binary; thus, no learning phase or expert knowledge is required. $\mathcal{CPE}$ enforces expected program behavior instead of defending against anticipated attacks; thus, new exploit variations will not circumvent $\mathcal{CPE}$. $\mathcal{CPE}$ is based on observable properties of the program counter and runtime callstack; a program has $\mathcal{CPE}$ if, for all program states during the execution of the program, the program counter and callstack are individually valid and consistent with each other. Code reuse attacks execute short sequences of instructions without respect to their location in original code; thus, these attacks deviate from our model.

Conformant program execution verifies each program state; therefore, continually validating it can result in high overhead. We address this problem with *observed conformant program execution* ($\mathcal{OCPE}$), which reduces overhead by only validating program state at system call executions. We demonstrate that this relaxed model is sufficient to detect code reuse attacks. Thus, $\mathcal{OCPE}$ provides an adoptable solution while still providing safety guarantees. $\mathcal{OCPE}$ is not designed to handle code reuse-based mimicry attacks, which have not yet been demonstrated in the research world or seen in the wild. We believe $\mathcal{OCPE}$ could be augmented in future work to handle these attacks.

We engineer our approach using a component model based on strong binary analysis of the code. This analysis allows us to operate on modern binaries, which frequently are highly optimized or lack debugging information; our analysis does not rely on information that may not be present in a modern application. We leverage a binary analysis toolkit to identify key characteristics of the stack frame at any instruction in the binary; this allows us to gather a full callstack via a *stackwalk* at runtime. While conceptually straightforward, accurate stackwalks are surprisingly difficult to perform on real applications. Our algorithm leverages these stackwalks, taken at system calls, to reliably detect code reuse attacks.

We implemented our code reuse detection algorithm in a tool, ROPStop, which operates on unmodified binaries, including running programs. We evaluated ROPStop using real exploits from two classes of code reuse attacks: return-oriented programming (ROP) and jump-oriented programming (JOP). Our results show that our tool is able to correctly identify each exploit. We tested ROPStop with the SPEC CPU2006 benchmarks and an Apache HTTP Server as a control group of unexploited, conventional binaries to evaluate overhead and measure the occurrence of false positives. Our results show an average overhead of 5.3% on and 6.3% on Apache; ROPStop reported no false positives.

We provide an overview of the challenges in detecting code reuse attacks and existing work in this area in Section 2 and a formal description of conformant

program execution and code reuse attacks in Section 3. Next, we describe the technical details of our approach in Section 4. We evaluate our approach in Section 5 and finish with a brief conclusion in Section 6.

2 Background and Related Work

Code reuse attacks provide interesting new challenges for security researchers. While $W \oplus X$ guards against code injection attacks, it is insufficient to stop code reuse attacks because such attacks do not write new code into the address space. We describe how an attacker gains control of the program and produces a code reuse attack. Further, we describe how an attacker locates gadgets within the program. We conclude with a discussion of techniques that are not specifically focused on code reuse attacks but are similar to techniques presented here.

2.1 Gaining Control of the Program

The first step of a code reuse attack is to gain control of the program counter to divert program control flow to the first gadget. This is done by making use of an existing vulnerability (e.g., a buffer overflow) to alter program data. Although these vulnerabilities and possible defenses are well studied, attackers remain able to exploit these vulnerabilities and launch attacks [51]. We assume that an attacker will be able to find a viable entry point for launching a code reuse attack, and do not discuss these vulnerabilities further. To ensure that program control flow will be diverted, an attacker overwrites either the return address for the calling function or a function pointer with the address of the first gadget. Note that $W \oplus X$ restricts attackers to cases where control flow targets depend on writeable locations rather than executable locations.

If the return address is overwritten, control flow will be diverted to the gadget when the current function returns and the new return address is loaded into the program counter. Stack-smashing attacks such as this one have been well-studied and techniques such as StackGuard [16] will prevent these attacks. To be effective, this protection must be present in *all program code*, including the program binary and any libraries on which it depends. These protections are provided as options by most modern compilers, but they are frequently not turned on by default, and can be turned off. Therefore, it is not safe to assume these protections will be present, and frequently it is not possible for a user to modify shared libraries (e.g., `libc.so`) to include them. If a function pointer is overwritten, control flow will be diverted to the gadget when the program invokes an indirect call or jump using the address of the function pointer.

Once control flow has been diverted to the first gadget, the code reuse attack begins. We note that an attack might also use an existing vulnerability to modify program data: for instance, to cause a system call to be executed with unintended arguments. These data driven attacks [2, 14, 20] are complementary to control flow-based attacks and are beyond the scope of our work.

2.2 Gadget Execution

Return- and Jump-Oriented Programming. In ROP, each gadget is terminated with a return instruction [44]. Thus, if an attacker has gained control of the stack pointer, they can use these return instructions to cause program execution to flow from one gadget to the next. JOP attacks use the same general technique, but gadgets are chained together with indirect jump instructions [7,9]. Thus, unlike ROP, JOP does not rely on manipulating the stack pointer; instead, indirect jump instructions have a specified target location, often stored in a register. This provides an extra challenge in constructing a jump-oriented attack: gadgets must manipulate relevant register values to ensure each indirect jump transfers control to the next gadget.

Defenses against Code Reuse Attacks. There are a variety of techniques designed to detect code reuse attacks. Several approaches make use of a shadow stack to prevent control flow manipulation that relies on overwritten stack values [12, 19, 23]. Others try to detect gadget execution by monitoring the length of instruction sequences between returns [11, 18]. Still others proposed monitoring pairs of call and return instructions [12, 31]. These approaches each target return-oriented attacks; however, they will not detect jump-oriented attacks because these attacks do not rely on return instructions to transfer control between gadgets. Another common approach ensures that a function should only start executing at its entry point [12, 29–31]. However, an attack may still hijack control flow while conforming to these requirements, thus remaining undetected.

Each of these mitigation techniques targets specific characteristics of previously observed code reuse attacks; such features may not be intrinsic to all code reuse attacks. In contrast, our work detects any violations of our model of conformant program execution by validating known properties of the program. Thus, evolving attack variations will not hinder our ability to detect these attacks.

Still other techniques monitor system calls to detect violations. ROPGuard [24] is a recent tool focused on ROP attacks that checks for a valid callstack at a subset of system calls. ROPGuard relies on frame pointers to traverse the callstack. Many modern programs do not save frame pointers; thus, callstack verification is turned off by default, leaving such programs vulnerable. By verifying conformant program behavior at *all* system calls and using a robust binary analysis toolkit to perform accurate stackwalks, our work provides a more complete approach.

2.3 Gadget Discovery

Locating potential gadgets is performed by scanning a target binary for return instructions (for ROP) or indirect jumps (for JOP). An attacker then chooses the gadgets to use from this set potential gadgets [13, 21, 47]. This selection of gadgets must allow the attacker to maintain command over the control flow of the program and perform desired actions while avoiding unwanted side effects.

ASLR is a common system-level technique that randomizes the addresses at which libraries are loaded into the address space of a process; this is particularly

relevant for code reuse attacks because these attacks rely on known locations for each gadget. Unfortunately, this is not sufficient to prevent a code reuse attack. Schwartz et al. point out that not all operating systems randomize all components within the address space, and some require an application to explicitly turn on ASLR [47]. As long as there exists some segment of code that is not randomized, code reuse attacks are possible. Checkoway et al. demonstrated that such attacks can be constructed even with a limited set of instructions [10]. Consequently, in real systems, we must assume that if gadgets exist in the code, an attacker will be able to find a sufficiently powerful set to perform their attack.

Several prevention techniques attempt to eliminate possible gadgets in library code via code diversification [17,28,32,36,53]. An alternative prevention strategy seeks to create binaries or kernels that lack necessary characteristics for ROP attacks [34, 35]. Many software diversification techniques rely on modifying the program, library, or kernel binaries via recompilation or binary rewriting; such modifications may not be possible in real systems. Furthermore, these techniques do not preclude code reuse attacks, but simply challenge attackers to identify gadgets in more sophisticated ways [51]. In contrast, ROPStop is engineered to provide a comprehensive solution that does not require ASLR or recompilation; ROPStop operates on unmodified binaries, including running programs.

2.4 Other Approaches

Our work is also similar to techniques that, while not focused on code reuse attacks, may be effective against such attacks: control flow validity enforcement and anomalous system call detection. Control-Flow Integrity (CFI) ensures that program execution holds to a control-flow graph (CFG) derived from static analysis [1]. However, because CFI verifies each control-flow transfer during program execution, it imposes high runtime overhead. More practical approaches, such as Control Flow Locking (CFL) [6], Compact Control Flow Integrity and Randomization (CCFIR) [54], and CFI for COTS binaries [55], have other limitations. CFL lazily verifies transfers, which greatly improves performance; however, their technique requires statically-linked binaries, which severely limits its application. CCFIR and CFI for COTS use binary rewriting to add verification checks at indirect control flow transfers. Although these approaches can be applied to shared libraries, protections must be applied to *all* binary code to ensure the implied security guarantees; any unprotected code is a potential attack target. Thus, the user applying protections must both be aware of and able to protect all library dependencies; this requirement can limit the applicability of these approaches. Further, CCFIR relies on ASLR for all program code. The limitations of these techniques prevent them from providing comprehensive defense solutions.

Host-based intrusion detection systems (IDS) use anomalous patterns of system calls to identify attacks. These approaches rely on a learning phase [22,25,48] or static binary analysis [26]. Unlike learning-based IDS, our work is based on a model of what program states are possible in normal execution. Further, our approach enforces a valid program state at each system call, rather than a valid pattern of system calls. We note that mimicry attacks [33, 42] allow an attacker

to subvert system call monitoring by ensuring that both the call stack of each system call and the sequence of system calls made by a compromised program appear normal to an IDS. However, mimicry attacks rely on code injection. While it is theoretically possible to extend a mimicry attack to employ code reuse, this form of attack does not appear in the wild; this would require an attacker to construct a sequence of gadgets that both executes their attack and restores the system to a state that appears valid at the next system call. However, gadget discovery is a difficult problem. In practice, attackers who employ code reuse are attempting to find the shortest route to a less restrictive environment [43]. Thus, we consider mimicry attacks beyond the scope of our work.

3 Conformant Program Execution

Our work is grounded in a model of conformant program execution. We create a definition of program state and then define requirements on that program state that must hold true at runtime. Program executions for which these requirements hold true are called *conformant* executions. Any deviation from these requirements during program execution indicates a non-conformant execution. We first describe our notation, then define a model of conformant program execution, and finally discuss how code reuse attacks will be detected as non-conformant.

3.1 Notation

A program is a set of procedures, $P = \{p_1, \ldots, p_m\}$. I is the set of all possible machine instructions; $I_P \subseteq I$ are the valid instructions for P. Each procedure p_j is a tuple, $p_j = (I_{p_j}, entry_{p_j}, \{exit_{p_j}\})$, where $entry_{p_j}$ and $exit_{p_j}$ are instructions in I_{p_j} that represent the entry point and the zero or more exit points for p_j. To represent valid interprocedural control flow in P, we define a call multigraph, $CMG_P = (N_P, E_P)$, where $N_P = P$ and each $e = (p_s, p_t, i) \in E_P$ is a control flow transfer $p_s \rightarrow p_t$ at i where $i \in I_{p_s}$ is the instruction that effects the call.

We parse the program binary using control and dataflow analysis; these analyses produce a call graph and control flow graph, which are used to populate the data structures in our model. More details are presented in Section 4. The resulting CMG_P may be incomplete due to unknown indirect control flow at $i \in p_j$. We address this incompleteness by making conservative assumptions about this control flow, such as that any procedure may be the target of an indirect call. In such cases, additional edges $(p_j, *, i)$ may be added to CMG_P, where $i \in p_j$ and the target of the control flow transfer at i is unknown. For increased accuracy, CMG_P could be augmented at runtime using dynamic analysis [46]; however, there is an increase in overhead associated with such analysis.

We define an execution of P as $execution(P) = \langle m_1, \ldots, m_n \rangle$, where each m_i represents an instance of program state. Program state includes elements of machine state that are affected by program execution, including registers and memory. We represent two elements of interest, the program counter and the callstack, with $pc(m)$ and $callstack(m)$. The callstack, C, is a sequence of

currently active stack frames, $C = \langle s_0, \ldots, s_n \rangle$, where $s_{top} = s_n$ is the frame at the top of the stack associated with the currently active procedure.

We refer to a stack frame as a tuple $s = (p, i, height)$, where p refers to the procedure associated with the stack frame. i refers to the last executed instruction in the context represented by s. For the top stack frame, this instruction is the program counter; for all other frames, this instruction is the call immediately before the frame's return address (Figure 1). *height* is the current height of the stack frame. Stack frame height is based on the space needed to store saved registers, procedure parameters, local variables, and a return address. Because different parts of a procedure may require different amounts of local storage, this height may vary for different parts of the procedure. Therefore the control flow path leading to the current location within a procedure, which we denote $entry_{proc(s)} \rightsquigarrow instr(s)$, affects the stack frame height. Our model does not rely on other information, such as a saved frame pointer, because such information may be omitted by optimizing compilers. We represent elements of each frame with $proc(s)$, $instr(s)$, and $height(s)$.

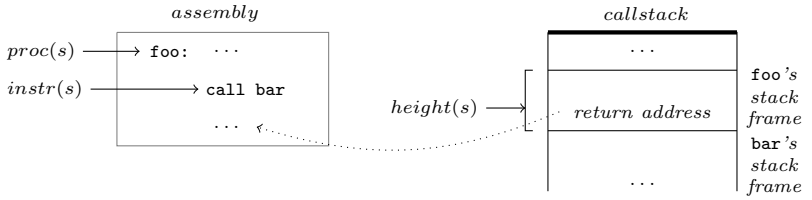

Fig. 1. Relationship between the tuple $s = (p, i, height)$ that represents the current stack frame instance for procedure `foo`, the assembly code for `foo`, and the callstack. Elements of the stack frame are represented with $proc(s)$, $instr(s)$, and $height(s)$. Note that the choice to associate the return address with the caller frame and not the callee frame is arbitrary; this could be restated without loss of generality.

We further define $Heights(i)$ to represent the set of *valid heights* possible from all paths through the CFG from $entry_{p_j}$ to $i \in p_j$. If code modifies stack frame height inside a loop, the height will depend on the number of loop iterations, so it is possible that the size of the set is not finite. In such cases, we assume that the stack frame height at these instructions is unknown. In principle, we could analyze the instruction sequence and build a closed form model of such behavior. In practice, this occurrence is exceedingly rare.

3.2 Conformant Program Execution

Our model of conformant program execution should be permissive enough to allow a program to execute valid program instructions reached via valid control flow, while restrictive enough to allow only these valid executions and detect code reuse attacks. We define a program to have conformant execution based on characteristics of two program state components: the program counter *pc*

and the runtime callstack C. A program is conformant at a given time, i.e., a particular machine state m_i, if each component is individually valid and if both components are consistent. A program has conformant program execution ($\mathcal{CPE}$) if the program is conformant for all program states in $execution(P)$.

To verify conformance for a particular program state during execution, we examine the program counter pc and the callstack C. The program counter should contain the address of a valid instruction. An instruction j is valid if it exists in the set of valid instructions for program P; $valid(i) : i \in I_P$. The program counter is valid if it points to a valid instruction; $valid(pc) : valid(instr(pc))$. This requirement eliminates the use of unaligned instructions that could provide a rich selection of unintended instruction sequences to be used in an attack [8]. Further, should $W \oplus X$ not be in place, this requirement precludes code injection attacks, which rely on code located outside of valid code sections of the binary.

A callstack $C = \langle s_0, \ldots, s_n \rangle$ is valid if a height requirement holds for each frame in C and if a call requirement holds for each pair of adjacent frames.

$$valid(C) : \forall\ s \in C : valid(s)$$

$$valid(s_k) : \begin{cases} valid\_height(s_k) & k = n \\ valid\_height(s_k) \wedge valid\_call(s_k) & 0 \leq k < n \end{cases}$$

A stack frame has a valid height h if that height is a member of the set of valid heights $Heights(i)$ for the corresponding instruction. For each pair of adjacent stack frames to meet the call requirement, the control transfer represented by each pair must correspond to an edge in the call multigraph.

$$valid\_height(s_k) :\ height(s_k) \in Heights(instr(s_k))$$

$$valid\_call(s_k) :\ (proc(s_k), proc(s_{k+1}), instr(s_k)) \in E_P$$

Validating calls between procedures associated with consecutive stack frames ensures that C represents a valid interprocedural control flow path through P. Thus, we incorporate the goals of CFI [1] but perform verification with runtime checks for an efficient implementation. Fratric also proposed a requirement on callstack validity [24]. In his work, a valid callstack requires each stack frame to have a valid return address and a valid call target, though the tool, ROPGuard, is only able to implement the former because it relies on information often not present in modern binaries. We discuss these technical differences in Section 4.

Finally, we define what is required for the program counter and the callstack to be consistent. Given a program counter that points to a valid instruction and a valid callstack, we need to ensure they are mutually consistent; that is, that the callstack is valid for that program counter: $consistent(i, C) : i = instr(top(C))$. These validity checks determine if a program has conformant execution:

$$valid(m) : valid(pc(m)) \wedge valid(callstack(m)) \wedge consistent(pc(m), callstack(m))$$

$$\mathcal{CPE}(P) : \forall\ m \in execution(P) : valid(m)$$

3.3 Code Reuse Attacks

We introduce a notation for code reuse attacks and discuss how our model detects these attacks. A code reuse attack is comprised of a series of gadgets,

$A = \langle g_0, \ldots, g_n \rangle$. Each gadget is a tuple: $g_k = (I_{g_k}, entry_{g_k}, exit_{g_k})$, where $entry_{g_k}$ and $exit_{g_k}$ are the instructions in I_{g_k} that represent the entry point and exit point for g_k. The instructions that comprise each gadget are part of the set of possible machine instructions I, but not necessarily part of I_P [49].

Although we provide a broad definition of gadgets that spans several flavors of code reuse attacks, there are several additional known characteristics of each gadget. Such characteristics are often used when crafting code reuse attacks [13, 47]. The last instruction in a gadget, $exit_{g_k}$, must cause a control flow transfer to the next gadget g_{k+1}. Gadgets g_k and g_{k+1} are chosen such that $exit_{g_k}$ transfers control to $entry_{g_{k+1}}$. In ROP, each $exit_{g_k}$ is a return instruction and the address of $entry_{g_{k+1}}$ is located at the top of the stack [49]; in JOP, $exit_{g_k}$ is an indirect jump instruction, where the target of the jump is $entry_{g_{k+1}}$ [7, 9].

Executing a sequence of gadgets interrupts the original execution of the program. However, gadget execution occurs in the context of the program and may use the original program stack. As a result, gadgets typically violate $\mathcal{CPE}$ in one or more of three ways: executing invalid instructions, altering the stack, or not following the original control flow of the program. To detect gadget execution, and thus a code reuse attack, we must identify these violations.

First, gadgets may execute invalid instructions, either code that was injected or misaligned instructions from the original program. We detect these invalid instructions by examining the program counter: $\forall\ g_k \in A : I_{g_k} \subseteq I_P$.

Second, gadgets may include instructions that alter the stack. Let $C = \langle s_0, \ldots, s_m \rangle$ represent the callstack prior to the execution of the first gadget, where s_m represents the top of the stack. While a gadget g_k executes, $C_k = \langle s_0, \ldots, s_n, \ldots, s_q \rangle$ where $\langle s_0, \ldots, s_{n-1} \rangle$ represents what remains of the original callstack and $\langle s_n, \ldots, s_q \rangle$ represents the effects of executing gadget code.

When g_0 begins execution, $s_n = s_m$. If subsequent gadgets remove values from the stack, $\langle s_0, \ldots, s_{n-1} \rangle \sqsubseteq \langle s_0, \ldots, s_m \rangle$, where $A \sqsubseteq B$ means that A is a contiguous subsequence of B that either starts at the same initial state (i.e., s_0) or is empty. Otherwise, $\langle s_0, \ldots, s_{n-1} \rangle = \langle s_0, \ldots, s_{m-1} \rangle$.

Each gadget must maintain the correct height in the context of the stack frame in which it executes; the expected height for the first instruction in each gadget, $entry_{g_k} \in I_{p_j}$, must match the height of s_n. Otherwise, the height of frame at the top of the stack will be observably incorrect at all instructions in the gadget. This invariant must be maintained as subsequent gadgets execute. We detect these invalid stack frames by verifying the height of each stack frame:

$$\forall\ i \in I_{g_k}, g_k \in A :\ height(entry_{g_k} \rightsquigarrow i) + height(s_n) = height(entry_{p_j} \rightsquigarrow i),$$
$$\text{where } I_{g_k} \in I_{p_j}$$

Third, gadgets may induce a control flow path that does not match that of the original program. The instructions that comprise each gadget are selected from an existing procedure; to maintain conformant execution, there must be a valid control flow transfer from the procedure corresponding to the stack frame below s_n to the procedure that contains the gadget's instructions. We detect these invalid control flow paths by validating the path represented by the callstack:

$$\forall\ g_k \in A : (proc(s_{n-1}), p_j, instr(s_{n-1})) \in E_P, \text{ where } I_{g_k} \in p_j$$

However, there are two ways in which an attack might be able to conform to our model. First, there may be cases in which a loop allows multiple valid stack heights at a particular instruction, as described in the previous section; in this case, a gadget may be able to use an invalid stack height without detection, i.e.,

$$|Heights(instr(s_k))| > 1 \ \wedge height(s_k) \in Heights(instr(s_k)) \ \wedge$$
$$height(s_k) \neq height(entry_{proc(s_k)} \rightsquigarrow instr(s_k))$$

In our experience, such code is rare, and can be mitigated by unrolling or otherwise modifying the loop such that $|Heights(instr(s_k))| = 1$. Second, an attack might take advantage of our conservative handling of indirect control flow to construct unintended paths through the binary via indirect procedure calls, i.e.,

$$(proc(s_{k-1}), *, instr(s_{k-1})) \in E_P \ \wedge \ proc(s_{k-1}) \not\rightarrow proc(s_k) \text{ at } instr(s_{k-1})$$

Such an attack would have to make use of existing program code using whole procedures at a time, because executing a sequence of procedure fragments would result in one or more invalid stack frames. Such attacks must still override the targets of one or more indirect calls to force the execution of the program down the path desired by the attack. For example, an attack could begin by overwriting a table of function pointers. However, this is made more complex by the fact that it must not only overwrite the function pointer but also overwrite the data used to set up the parameters used at the callsite. While this strategy seems unlikely, as we use more sophisticated analysis to refine indirect call edges, we further reduce the likelihood of circumventing our technique.

3.4 Observed Conformant Program Execution

In this section, we define observed conformant program execution ($\mathcal{OCPE}$), which improves efficiency by verifying program states only at system call entries. Monitoring at system call granularity offers two advantages. First, the system call tracing interface provided by operating systems (e.g., `PTRACE_SYSCALL`) cannot be interrupted; therefore, we will receive notification of system call entry even if an attack is in progress. Second, attacks must use the system call interface (e.g., `exec`) to modify the overall machine state; therefore, we will not miss the effects of an attack by monitoring only at system calls. We define $\mathcal{OCPE}$ as follows:

$$is\_syscall(m) : callstack(m) = \langle s_0, \ldots, s_n \rangle; proc(s_n) \text{ invoked system call } syscall(m)$$
$$observed(P) = \langle m \in execution(P) \mid is\_syscall(m) \rangle$$
$$\mathcal{OCPE}(P) : \ \forall \ m \in observed(P) : \ valid(m)$$

$\mathcal{OCPE}$ makes the assumption that the effects of an attack will be visible in the program state at the point a system call is executed. As discussed in Section 2, current code reuse attacks are not constructed to evade $\mathcal{OCPE}$ [13,47]. We demonstrate in Section 5 that $\mathcal{OCPE}$ is effective against these attacks. Furthermore, we believe it will be difficult for future code reuse attacks to entirely hide their effects from our stack model, because an attack would have to both hide its own effects as well as forge a consistent program state. Even if a future attack could circumvent $\mathcal{OCPE}$ in this way, our model of $\mathcal{CPE}$ would detect these deviations from normal program execution. Therefore, $\mathcal{OCPE}$ is an effective optimization of $\mathcal{CPE}$ that greatly improves performance while preserving the power of $\mathcal{CPE}$.

4 Implementation

We have incorporated our model into a tool, ROPStop, that monitors a process during its execution for observed conformant program execution. We use several components from the Dyninst binary modification and analysis toolkit to perform this runtime monitoring and verification [37], and ROPStop has been implemented for both 32- and 64-bit x86/Linux. In total, ROPStop is approximately 3,000 lines of C++ on top of several toolkit libraries.

4.1 Process Monitoring

We perform runtime process monitoring using the ProcControlAPI process monitoring and control component [40] of Dyninst. This library allows a tool process (ROPStop) to manage one or more target processes using platform-independent abstractions. ProcControlAPI includes the ability to control threads, set breakpoints, request notifications (callbacks) at events, and read and write memory.

Using ProcControlAPI, we may either create a new process or attach to a running process. We then register a callback function that is invoked before each system call. Although ProcControlAPI already provides the capability to register callback functions at various process events using the operating system's debug interface (`ptrace`), we extended the library to provide support for callbacks at system call entry. ROPGuard [24] used an alternative approach that instead operates on the library wrappers for system calls rather than directly on the system call. However, this can be defeated by malicious code that directly executes a trap instruction to invoke a system calls By using the debug interface, we can guarantee that all system calls will trigger our callback.

ROPStop then parses the program binary and any library dependencies using the ParseAPI control flow analysis library [39]. ParseAPI uses recursive traversal parsing to construct a whole-program control flow graph [15, 52], including sophisticated heuristics to recognize functions that are only reached by indirect control flow and works in the absence of symbol table information [27, 45]. This CFG provides both the call multigraph (CMG_P) required by our model and the information necessary to identify each valid program instruction (I_P).

Once the process binary has been parsed, we continue its execution. If the process creates additional threads or launches a new process, ProcControlAPI will monitor these also. A monitored process is stopped at each system call entry. ROPStop checks that the program has conformant execution; ROPStop explicitly verifies the program counter and the callstack and implicitly verifies consistency. If the process is conformant, execution is continued. If ROPStop detects non-conformant program state, the process is terminated.

4.2 Instruction Validity

ROPStop first verifies that the program counter points to a valid instruction in the original program, e.g., that $instr(pc) \in I_P$. This inexpensive step ensures that an attack may not make use of unaligned instructions. We identify the basic

block in the CFG that contains this address and disassemble the block using the InstructionAPI instruction disassembly library [38]. If we reach an instruction that begins at the address in the program counter, we conclude the current instruction is valid. Otherwise, we conclude the instruction is invalid.

4.3 Callstack Validity

Next, ROPStop gathers a full stackwalk using the StackwalkerAPI library [41]. A full stackwalk must be gathered at each system call to ensure that no malicious stack modifications have occurred since the last validity check; this step must be completed even if two system calls occur within the same function or a single loop. ROPStop uses StackwalkerAPI to walk the stack one frame at a time and validate each frame before continuing the stackwalk. StackwalkerAPI represents stack frames as pairs (r, sp), where r is a location in the program and sp is the value of the stack pointer in that location. Given an input frame (r_i, sp_i), StackwalkerAPI calculates the expected previous frame (r_{i-1}, s_{i-1}).

Given a valid stack frame s_j and an expected previous frame s_{j-1} from StackwalkerAPI, we validate it as follows. We verify the height of the stack frame s_j by validating the return address of s_{j-1}, r_{j-1}. We define a return address to be valid if the instruction at this address is immediately preceded by a call instruction; this definition is also used by the ROPGuard tool [24]. ROPStop uses InstructionAPI to locate the instruction prior to the return address and identify its type. If s_{j-1}'s return address is valid, we conclude that the height of s_j is valid, e.g., that $height(s_j) \in Heights(instr(s_j))$.

This validation allows ROPStop to verify the second necessary condition for a valid stack frame: that there exists a valid control flow transfer between the caller (s_{j-1}) and the current, callee frame (s_j). We check the CFG constructed by ParseAPI to verify that there is a valid call edge from the caller to the callee; this edge must originate at the call instruction found in the previous step. This verifies that $(proc(s_{j-1}), proc(s_j), instr(s_{j-1})) \in E_P$.

Performing a stackwalk is not always an easy task. The debugging information that describes stack frames (e.g., DWARF call frame information) is frequently missing; for example, commercial binaries frequently omit this information due to concerns about reverse engineering. Even if this debug information is present it is frequently incomplete or incorrect; for example, compilers often omit stackwalking information for automatically generated code.

In light of these challenges, we extended StackwalkerAPI to use dataflow analysis to identify stack heights. This analysis begins at the entry to a function and tracks the effects of all instructions that modify the stack heights; the result is the set of possible stack heights for each instruction in the function. This robust analysis enables an accurate stackwalk in the absence of debugging information.

If StackwalkerAPI reaches the bottom of the stack and does not encounter an invalid stack frame, then we conclude that the callstack is valid. Otherwise, we have found non-conformant program state, and the process is terminated.

5 Evaluation

ROPStop provides protection against code reuse attacks while identifying no false positives. We verified these characteristics with the following experiments. First, we tested ROPStop against code reuse attacks, as well as a conventional stack smashing attack, and show that ROPStop detects each of these attacks. Second, we tested ROPStop against a set of conventional binaries, and show that ROPStop results in no false positives while imposing overhead of only 5.3% on SPEC benchmarks and 6.3% on an Apache HTTP Server.

All evaluation was conducted on a 2.27GHz quad-core Intel Xeon with 6GB RAM, running RHEL Server 6.3 (kernel 2.6.32). All exploits were run inside VirtualBox 4.2.0 virtual machines, running Debian 5.0.10 (2.6.32) or Ubuntu 10.04 (2.6.26); see Table 1. SPEC and Apache were run directly on the host.

Table 1. Details about each exploit and results for ROPStop's detection of the attack. All attacks were detected because of invalid stack frame heights; the exploit characteristic that lead to the observed invalid program state is also provided.

Name	OS	Exploit Source	Detected	Why Invalid
17286a (ROP)	Ubuntu 10.04	sickness [50]	✓	Overwritten return address
17286b (ROP)	Ubuntu 10.04	sickness [50]	✓	Overwritten return address
Rsync (ROP)	Debian 5.0.10	Schwartz et al. [47]	✓	Overwritten return address
Bletsch (JOP)	Debian 5.0.10	Bletsch et al. [7]	✓	Gadget executing
Stack-smash	Debian 5.0.10	Aleph One [3]	✓	Overwritten return address

We began by testing ROPStop against known attacks; we summarize each attack in Table 1. The 17286 ROP exploits are from `http://www.exploit-db.com`. Rsync is a ROP exploit generated by the Q tool [47]. We acquired the code necessary for this exploit from the authors; additional exploits from this work were unavailable. Bletsch is a JOP attack [7]. Finally, we include a canonical buffer-overflow attack, Stack-smash, to demonstrate ROPStop's ability to detect these attacks also. We used information provided in the original documentation to create the vulnerable program as well as the input necessary for each exploit.

The results of testing ROPStop against these attacks are shown in Table 1. We were successful in detecting each attack. In each case, callstack verification failed; the height of a stack frame was found to be invalid because the return address in the expected caller frame did not follow a `call` instruction.

Next, we used the SPEC CPU2006 benchmark suite and an Apache HTTP Server (version 2.4.6) [5] to provide a control group of conventional binaries. We applied ROPStop to the execution of these binaries to both detect any false positives and determine how much overhead ROPStop incurs. We selected SPEC because the execution of each benchmark is well understood and as CPU intensive programs, any overhead imposed by ROPStop would not be hidden by I/O. Each benchmark was run three times with the reference inputs; we report the mean of these runs. We measured end-to-end times, which includes the time required to generate our model as well as the runtime of the program.

ROPStop generated no false positives when run on the SPEC CPU2006 benchmark suite. The overhead imposed by ROPStop is shown in Figure 2; on average,

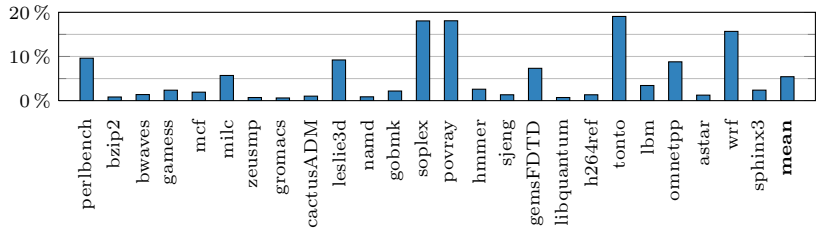

Fig. 2. Overhead results for SPEC CPU2006 benchmarks under ROPStop; **mean** represents the geometric mean of all overhead values. We omit four benchmarks, `gcc`, `calculix`, `dealII`, `xalancbmk`; we were unable to successfully run these unmonitored.

ROPStop imposes 5.3% overhead. In most cases ROPStop imposes under 3% overhead; our highest overhead is 19.1% for `tonto`.

Table 2. Full results for SPEC CPU2006 benchmarks under ROPStop. The system call rate is reported as system calls per second based on unmonitored runtimes; components of overhead are reported as percentages of total overhead imposed (summing to 100%).

			Overhead Breakdown		
Benchmark	System Call Rate (calls/second)	% Overhead Imposed	% Instruction Validity	% Callstack Validity	% Context Switching
perlbench	167.8	9.6	0.2	50.7	49.1
bzip2	2.0	0.8	0.0	43.8	56.2
bwaves	3.3	1.4	0.0	31.7	68.2
gamess	29.5	2.4	0.1	59.0	40.9
mcf	3.4	1.9	0.0	51.4	48.5
milc	25.5	5.7	0.1	23.5	76.4
zeusmp	0.2	0.7	0.0	54.2	45.7
gromacs	1.5	0.6	0.0	45.1	54.9
cactusADM	7.6	1.0	0.1	52.0	47.9
leslie3d	31.4	9.2	0.1	14.2	85.7
namd	3.1	0.9	0.0	61.6	38.4
gobmk	14.0	2.2	0.1	43.6	56.4
soplex	241.5	18.1	0.2	50.8	49.1
povray	156.2	18.1	0.1	53.3	46.6
hmmer	18.6	2.6	0.1	38.8	61.1
sjeng	4.8	1.3	0.0	40.5	59.5
GemsFDTD	88.7	7.3	0.2	40.9	59.0
libquantum	0.3	0.7	0.0	50.2	49.8
h264ref	5.1	1.3	0.0	28.9	71.0
tonto	119.6	19.1	0.1	41.2	58.7
lbm	1.4	3.4	0.0	15.5	84.5
omnetpp	3.7	8.8	0.0	12.7	87.3
astar	7.2	1.3	0.1	56.7	43.3
wrf	53.2	15.7	0.0	13.4	86.6
sphinx3	18.6	2.4	0.1	38.1	61.8

The overhead imposed by ROPStop is dependent on the frequency of system calls as well as the height of the stack. ROPStop is a separate process, rather than in the same address space as the monitored application. Validating state at each system call requires at least two context switches; these context switches are more expensive than the validation checks ROPStop performs. Thus, benchmarks that make frequent use of system calls or, to a lesser extent, have deep call stacks,

suffer higher overhead. We report the breakdown of this overhead in Table 2. The SPEC benchmarks vary greatly in the number of invoked system calls. The average number of system calls is 23,635; `zeusmp` invokes only 140, while `tonto` invokes 153,890 system calls. We omit four benchmarks, `gcc`, `calculix`, `dealII`, and `xalancbmk`, because we were unable to run them unmonitored; we expect these to have similar performance to our reported numbers.

We used the ApacheBench tool [4] to measure the performance of the Apache server while being protected by ROPStop. We ran ApacheBench with various total numbers of requests to a local, static page; the number of concurrent requests was always 100. Each request size was run twenty times; we report the mean of these runs. For each run, we started the server, attached ROPStop, and then ran ApacheBench. We report numbers recorded by ApacheBench, rather than end-to-end results, because web servers are commonly long-running applications; however, model generation times were similar to those for SPEC. ROPStop generated no false positives when run on the Apache server. The overhead imposed is shown in Figure 3; on average, ROPStop imposes 6.3% overhead. The Apache server made between approximately 3,000 and 60,000 system calls for the smallest and largest total number of requests, respectively.

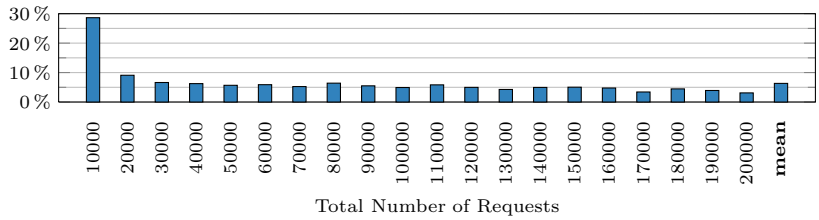

Fig. 3. Overhead results for an Apache HTTPD web server run under ROPStop, measured using ApacheBench; **mean** represents the geometric mean of all overhead values.

6 Conclusion

We have presented a systematic approach for detecting code reuse attacks. In contrast to current techniques, which rely on exploit characteristics that may not be intrinsic to the class of attacks, our approach is based on first principles. We defined a model of conformant program execution; by verifying the program counter and callstack, we were able to detect code reuse attacks, even as they continued to evolve. To provide an efficient and adoptable solution, we also defined observed conformant program execution, which provided the same guarantees while only verifying program conformance at system calls. Finally, we built a tool, ROPStop, that uses our model of observed conformant program execution to detect code reuse attacks. Our results show that ROPStop is capable of accurately detecting real code reuse attacks. Further, when tested on a set of modern applications, ROPStop produced no false positives and incurred only 5.3% overhead on SPEC CPU2006 and 6.3% on an Apache HTTP Server.

Acknowledgments. This work is supported in part by Department of Energy grants DE-SC0003922 and DE-SC0002154; National Science Foundation Cyber Infrastructure grants OCI-1032341, OCI-1032732, and OCI-1127210; and Department of Homeland Security under AFRL Contract FA8750-12-2-0289.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Info. & Systems Security (TISSEC) 13, 4:1–4:40 (2009)
2. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing Memory Error Exploits with WIT. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2008)
3. Aleph One: Smashing the stack for fun and profit. Phrack Magazine 7(49), 14–16 (1996)
4. Apache Software Foundation: ab - Apache HTTP server benchmarking tool (July 2013), `http://httpd.apache.org/docs/2.2/programs/ab.html`
5. Apache Software Foundation: Apache HTTP Server Project (July 2013), `http://www.apache.org`
6. Bletsch, T., Jiang, X., Freeh, V.: Mitigating Code-Reuse Attacks with Control-flow Locking. In: Annual Computer Security Applications Conference (ACSAC), Orlando, FL (December 2011)
7. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: ASIACCS, Hong Kong, China (March 2011)
8. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In: ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (October 2008)
9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-Oriented Programming without Returns. In: ACM Conference on Computer and Communications Security (CCS), Chicago, IL (October 2010)
10. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. In: EVT/WOTE, Montreal, Canada (August 2009)
11. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: Detecting Return-Oriented Programming Malicious Code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009)
12. Chen, P., Xing, X., Han, H., Mao, B., Xie, L.: Efficient Detection of the Return-Oriented Programming Malicious Code. In: Jha, S., Mathuria, A. (eds.) ICISS 2010. LNCS, vol. 6503, pp. 140–155. Springer, Heidelberg (2010)
13. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., Yin, X.: Automatic construction of jump-oriented programming shellcode (on the x86). In: ASIACCS, Hong Kong, China (March 2011)
14. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium, Baltimore, MD (July 2005)
15. Cifuentes, C., Van Emmerik, M.: UQBT: Adaptable Binary Translation at Low Cost. Computer 33(3), 60–66 (2000)
16. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: USENIX Security Symposium, San Antonio, TX (January 1998)

17. Davi, L., Dmitrienko, A., Nurnberger, S., Sadeghi, A.R.: Gadge Me If You Can: Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In: ASIACCS, Hangzhou, China (May 2013)
18. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: ACM Workshop on Scalable Trusted Computing (STC), Chicago, IL (November 2009)
19. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In: ASIACCS, Hong Kong, China (March 2011)
20. Demay, J.C., Majorczyk, F., Totel, E., Tronel, F.: Detecting illegal system calls using a data-oriented detection model. In: International Information Security Conference (SEC), Lucerne, Switzerland (June 2011)
21. Dullien, T., Kornau, T., Weinman, R.P.: A framework for automated architecture-independent gadget search. In: USENIX Workshop on Offensive Technologies (WOOT), Washington, D.C. (August 2010)
22. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2003)
23. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: ACM Workshop on Secure Execution of Untrusted Code (SecuCode), Chicago, IL (November 2009)
24. Fratric, I.: ropguard (2012), `http://code.google.com/p/ropguard/`
25. Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: ACM Conference on Computer and Communications Security (CCS), Washington, D.C. (October 2004)
26. Giffin, J.T., Jha, S., Miller, B.P.: Automated Discovery of Mimicry Attacks. In: Zamboni, D., Kruegel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 41–60. Springer, Heidelberg (2006)
27. Harris, L.C., Miller, B.P.: Practical Analysis for Stripped Binary Code. ACM SIGARCH Computer Architecture News 33(5), 63–68 (2005)
28. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: Where'd My Gadgets Go? In: IEEE Symposium on Security and Privacy, San Francisco, CA (May 2012)
29. Huang, Z., Zheng, T., Liu, J.: A Dynamic Detective Method against ROP Attack on ARM Platform. In: International Workshop on Software Engineering for Embedded Systems (SEES), Zurich, Switzerland (June 2012)
30. Huang, Z., Zheng, T., Shi, Y., Li, A.: A Dynamic Detection Method against ROP and JOP. In: International Conference on Systems and Informatics, Yantai, China (May 2012)
31. Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Branch Regulation: Low-Overhead Protection from Code Reuse Attacks. In: International Symposium on Computer Architecture (ISCA), Portland, OR (June 2012)
32. Kemerlis, V.P., Portokalidis, G., Keromytis, A.: KGuard: Lighweight Kernel Protection against Return-to-user Attacks. In: USENIX Security Symposium, Bellevue, WA (August 2012)
33. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: USENIX Security Symposium, Baltimore, MD (July 2005)
34. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating Return-Oriented Rootkits with "Return-Less" Kernels. In: European Conference on Computer Systems (EuroSys), Paris, France (April 2010)

35. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-Free: Defeating Return-Oriented Programming Through Gadget-less Binaries. In: Annual Computer Security Applications Conference (ACSAC), Austin, TX (December 2010)
36. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In: IEEE Symposium on Security and Privacy, San Francisco, CA (May 2012)
37. Paradyn Project: Dyninst (2013), http://www.dyninst.org
38. Paradyn Project: InstructionAPI (2013), http://www.dyninst.org
39. Paradyn Project: ParseAPI (2013), http://www.dyninst.org
40. Paradyn Project: ProcControlAPI (2013), http://www.dyninst.org
41. Paradyn Project: StackwalkerAPI (2013), http://www.dyninst.org
42. Parampalli, C., Sekar, R., Johnson, R.: A practical mimicry attack against powerful system-call monitors. In: ASIACCS, Tokyo, Japan (March 2008)
43. Polychronakis, M., Keromytis, A.: ROP Payload Detection Using Speculative Code Execution. In: International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico (October 2011)
44. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans. Info. & Systems Security (TISSEC) 15(1), 2:1–2:34 (2012)
45. Rosenblum, N., Zhu, X., Miller, B., Hunt, K.: Learning to Analyze Binary Computer Code. In: AAAI, Chicago, IL (2008)
46. Roundy, K.A., Miller, B.P.: Hybrid Analysis and Control of Malware Binaries. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 317–338. Springer, Heidelberg (2010)
47. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit Hardening Made Easy. In: USENIX Security Symposium. San Francisco, CA (August 2011)
48. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2001)
49. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security (CCS), Alexandria, VA (October 2007)
50. sickness: Linux exploit development part 4 - ASCII armor bypass + return-to-plt (2011), http://sickness.tor.hu/?p=378
51. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: IEEE Symposium on Security and Privacy (May 2013)
52. Theiling, H.: Extracting safe and precise control flow from binaries. In: Conference on Real-Time Computing Systems and Applications, Washington, D.C. (December 2000)
53. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In: ACM Conference on Computer and Communications Security (CCS), Raleigh, NC (October 2012)
54. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical Control Flow Integrity & Randomization for Binary Executables. In: IEEE Symposium on Security and Privacy, San Francisco, CA (May 2013)
55. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: USENIX Security Symposium, Washington, D.C. (August 2013)