Track Conventions, Not Attack Signatures: Fortifying X86 ABI and System Call Interfaces to Mitigate Code Reuse Attacks

Sarp Ozdemir, Rutvik Saptarshi, Aravind Prakash and Dmitry Ponomarev
{sozdemi2, rsaptar2, aprakash, dponomar}@binghamton.edu
Binghamton University, NY

Abstract

Code Reuse Attacks (CRAs) are dangerous exploitation strategies that allow attackers to compose malicious programs out of existing application and library code gadgets, without requiring code injection. Previously, researchers explored hardware-assisted protection schemes that track attack signatures to identify malicious behavior. This paper makes two main contributions. First, we show that previously proposed signature-based schemes are impractical because they do not always distinguish attack patterns from the behavior of benign programs. Second, we demonstrate that instead of tracking attack signatures, a more robust defense mechanism is to track legitimate usage of system calls and ABI compliance in hardware, and detect deviations from established conventions as possible attacks. We propose two specific tracking mechanisms: the setting of arguments for system calls and register usage across function calls. We demonstrate that our solution severely hinders practical CRAs and completely stops code-reuse execution of sensitive system calls like `mprotect`*. Our solution imposes very low performance overhead and modest design complexity.*

1. Introduction

Code Reuse Attacks (CRAs) are a dangerous exploitation method in computer systems [7, 30, 49]. The core idea of a CRA is to compose a malicious program by stitching together pieces of existing code, called *gadgets*, and controlling transition between the gadgets using indirect branch instructions. Since no new code is injected by an attacker, defenses that disallow execution from writable memory, such as Data Execution Prevention (DEP) [39], are not effective against CRAs. CRAs come in many forms, ranging in complexity from basic *return-to-libc* attacks [52], to Return-oriented Programming (ROP) [10, 49] and Jump-oriented Programming (JOP) [7, 15], to more complex Counterfeit Object-Oriented Programming (COOP) [48], Block-Oriented Programming (BOP) [28], Function-Oriented Programming (FOP) [25] and Printf-Oriented Programming (POP) [13]. Recently, CRA attacks that target SGX enclaves have also been proposed [6]. Although Control-Flow Integrity (CFI) [1] is known to be principled and promising in mitigating control-flow hijacking attacks, in practice, they are known to be imperfect [34], and modern attacks can evade them [29, 48].

Many different approaches for detecting CRAs have been proposed. On the one hand, from a policy perspective, the key question is whether the focus of detection should be: a) on specific attack patterns (signatures), or b) deviations from normal program behavior? But on the other hand, from an enforcement perspective, one needs to decide whether defenses must be deployed at the software level or the hardware level or both. Historically, hardware solutions are known to be highly effective, robust (e.g., DEP, W $\bigoplus$ X) and impose low performance overhead. They are transparent to the software layers and provide full-system protection. However, unlike software-level defenses, they are not easily configurable and offer deployment challenges. If hardware solutions provide a high level of flexibility and scalability, they are clearly a favorite choice.

Several previous works advocated detection of attack signatures, including techniques that use hardware support [31, 32]. In particular, these defenses rely on *gadget length* and the number of gadgets that are executed consecutively as attack indicators. In general, there are two potential problems with such signature-based schemes. First, it is possible that if the details of the defense are known, the adversary can attempt to modify the attack to bypass the protection. Earlier work [23] has shown that ad hoc parameter values used in past defenses [17, 43] can be bypassed, and highlighted the difficulty of choosing the values of these parameters. Second, and perhaps even more importantly, signature-based schemes can lead to a large amount of false positives, where legitimate application code will be flagged as an attack. In this paper, we demonstrate (Section 3) several practical examples of programs that legitimately experience gadget-like behavior and are therefore truly indistinguishable from attacks in terms of consecutive number of gadgets and gadget lengths. False positives make the defense less practical, as users are likely to turn off the defense. These results challenge the viability of signature-based detection schemes. Formally establishing this conclusion is the *first contribution of this paper*.

Instead of tracking signatures, we argue that a more effective approach to CRA detection is to track deviations from normal program behavior, specifically deviations from *execution conventions* of benign programs. Particularly, we make two key observations: (a) indirect branching is in the heart of code-reuse attacks and (b) because sensitive system call invocation (e.g., `mprotect`) is often the goal of CRAs, protecting system call interface is critical to defense.

With these observations, we propose a novel CRA-centric system call defense. We observe that in typical benign code, all (or most) of the system call arguments are set together right before executing a syscall instruction, usually within the same basic block. In contrast, in a CRA, system call arguments are set one-at-a-time, each argument in its own separate gadget. This disparity offers a new detection opportunity by monitoring the number of indirect branch instructions between the setting of system call arguments and the invocation of the syscall instruction. CRA defense based on this observation is the *second contribution of this paper*.

We present a light-weight hardware design to track the violation of system call argument setting. We demonstrate that our proposed defense is simple and effective. Specifically, we show that our defense can completely prevent code-reuse execution of system calls. Our solution incurs minimum performance overhead and incurs about 150 bytes of on-chip storage (although this number depends on the number of system calls tracked). A major advantage of our technique is that it is program-agnostic and it does not rely on any specific characteristics of programs. Additionally, because existing software stack already adheres to the underlying ABI, *our solution requires minimal*[1] *changes to the OS, compiler and the program binary*, and thus *offers backward compatibility*.

The rest of the paper is organized as follows. Section 2 presents the background on code reuse attacks. Section 3 describes the limitations of existing signature-based schemes and shows (for the first time) specific code patterns in real programs that would cause signature schemes to generate false positives. Section 4 presents a high-level overview of our design, Section 5 describes the system operation and design, and Section 6 describes microarchitectural support. We present performance and security evaluation in Section 7, review related work in Section 8 and offer our concluding remarks in Section 9.

2. Technical Background

2.1. CRA Example

We demonstrate an of example jump-oriented programming attack that was crafted using *libc.so.6* as our code base. We used the gadget discovery algorithm proposed in [7,49] and rewrote the algorithm for x86-64 using the Capstone library [12]. JOP attacks use special dispatcher gadgets to connect functional gadgets without using returns. To search for dispatcher gadgets from the gadget pool, we used the dispatcher discovery algorithm proposed in [31]. One of the dispatcher gadgets found in *libc* is shown in Figure 1.

```
pop rsi
jmp qword ptr [rsi + 0x41]
```

Figure 1: A dispatcher gadget from libc.

[1]In order to support exception handling, multi-threading and policy configuration.

This dispatcher gadget uses register `rsi` as the *gadget program counter* (GPC), which holds the starting address of the next functional gadget. The `pop` instruction updates and loads the GPC into the `rsi` register. The attacker must ensure that the calculated address `rsi + 0x41` points to a location in their payload that contains the address of the next gadget.

The attack goal is to make a system call with the `sys_execve` function, which launches a new shell. Arguments must be passed into this system call as follows: `sys_execve("/bin/sh", ["/bin/sh"], NULL)`. The `rdi`, `rsi`, and `rdx` registers must contain the address of `"/bin/sh"`, the address of the arguments array `["/bin/sh"]` terminated by a null pointer, and a null pointer, respectively. The `rax` register must contain the system call number `0x3b` before executing the `syscall` instruction.

We used six functional gadgets, all found within the *libc* codebase, to implement this attack. We refer to these gadgets by their corresponding number shown in the leftmost column of Figure 2.

```
G1 | pop rcx        ; set dispatcher address
   | sal bl, 1      ; not used
   | jmp rcx        ; return to dispatcher
---------------------------------------------------
G2 | pop rax        ; set dispatcher address
   | jmp rcx        ; return to dispatcher
---------------------------------------------------
G3 | pop rdi        ; set address of filename
   | xor rbx, rbx   ; irrelevant instruction
   | jmp rax        ; return to dispatcher
---------------------------------------------------
G4 | pop rcx        ; set location of disp.  addr
   | jmp rax        ; return to dispatcher
---------------------------------------------------
G5 | pop rdx        ; set null ptr address
   | jmp [rcx]      ; return to dispatcher
---------------------------------------------------
G6 | mov eax, 0x3b  ; set syscall number
   | syscall        ; call execve
```

Figure 2: Functional gadgets used in the example attack

To commence the attack, we set a register to the dispatcher address using gadget `G1`. Next, register `rdi` is set to the address of the string `"/bin/sh"`. We used gadget `G2` to set `rax` to the dispatcher address. Then, gadget `G3` can be used to set `rdi` to the string address.

Register `rdx` needs to contain the null pointer. We found a gadget `G5` that performed a *memory* indirect branch with `rcx`. Since `rcx` would not contain the appropriate value for a memory indirect branch, we overwrite `rcx` with `G4`. With `G5`, we load the null pointer into `rdx`. We left `rsi` and `rax` toward the end of the attack because these registers were constantly overwritten by the dispatcher gadget and the functional gadgets. `rsi` needs to point to an array containing the address of the string `"/bin/sh"` followed by a null pointer. We crafted our payload such that the value loaded into `rsi` for the final time was the address of `argv[]`. We also ensured that the calculated address `rsi + 0x41` points to a location containing the address of our final gadget `G6`. By the time the attacker branches to `G6`, register `rsi` contains the address of `argv[]`. Finally, gadget `G6` loads the system call number into register rax and completes the attack with the call to execve.

2.2. CFI-Evading Attacks

Control-flow integrity (CFI) is a popular well-studied defense against code-reuse attacks [1,51,53]. Consider the example in Figure 3. Since function f2 is invoked using a function pointer, the compiler can not reason about the target at compile time, as static control-flow integrity must allow any indirectly invoked functions whose addresses are referenced as valid targets.

However, a modern CRA in Figure 3 takes advantage of such an over-approximated CFI policy. By transferring to possible targets of indirect branches (i.e., entry point of address-taken functions (gadgets **EG1, EG2**) and/or locations where return instructions can return to (e.g., call-preceded **CP1** gadget), the attacker can achieve subversion by evading CFI.

3. Limitations of the State-of-the-Art Defenses

Signature-based hardware defenses Previous works on signature-based detection [17,31] view programs with a narrow lens of gadget length and gadget count, and flag many benign applications that exhibit CRA-like execution as attacks. In Figure 4, we show an example from an application called BAP [9], which is written in OCaml. This application frequently executes repetitive, short snippets of code that are separated by indirect branches. Each line of code in Figure 4 shows the indirect branch instruction, and the number of instructions that preceded the indirect branch. This benign code pattern would be flagged as an attack by previous defenses, as the number of instructions between indirect branches are small and the number of consecutive gadgets are large. The Figure 5 shows even shorter gadget lengths that naturally occur during program execution. The instruction trace from Figure 5 was found in Pandoc [37].

We also found that method overloading implementation in Objective C (see Figure 6) programs mimic CRA-like behavior. In essence, _objc_msgSend is an Objective C subroutine that is called before *every* method invocation [2]. The _objc_msgSend subroutine is a way for Objective C objects to call its methods. As this subroutine is called very frequently, it is written in assembly code for minimal performance overhead [3,55], and thus appears to look like a CRA attack with a gadget length of 13 instructions. The example in Figure 6 shows that the _objc_msgSend subroutine is called before the allowsVibrancy method of an NSAppearance object.

Another weakness of using gadget length and gadget count thresholds are that the thresholds as defense heuristics may not be effective to newer applications in the future. This creates the need to occasionally update the defense parameters. For example, we tested the CRA defense called SCRAP [31] which tracks JOP attack signatures using gadget counts and lengths on the newer SPEC 2017 benchmarks, and found many false positives under the original proposed thresholds. Loosening these thresholds to decrease the number of false positives would consequently make it easier for attackers to execute an attack [23].

These examples from real programs demonstrate that it is very difficult, if not impossible, to detect CRAs based on attack signatures without creating a significant number of false alarms.

CFI-based defenses Multiple solutions both at software- and hardware-levels have attempted to enforce CFI as a defense primitive. Additionally, shadow-stack based defenses have been deployed to prevent return-address corruption. While these defenses have been effective in handling simple CRAs, modern attacks such as Control-flow Bending [14], Block-Oriented Programming [28], COOP [48] and CCFIR [22] evade CFI-based defenses by operating within a statically recoverable CFG. They leverage high-level program semantics such as the printf format string [14], C++ virtual function dispatch [48] and function trace-level uncertainties [29] that are hard to recover in the hardware.

4. Our Approach

4.1. Threat Model

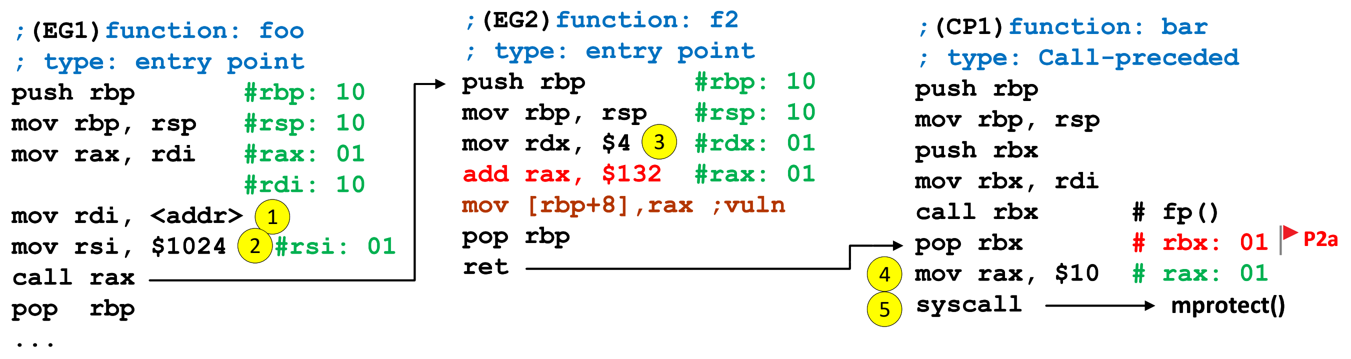
We address a threat model not unlike other defenses against code reuse attacks. We assume an execution stack where the kernel and the hardware are uncompromised and trusted. Additionally, the underlying system software, i.e., compiler and the dynamic linker/loader are trusted, and the hardware is capable of preventing data execution (e.g., NX). Further, we assume that a potential attacker has access to the application binary and is able to identify and chain gadgets to construct a CRA. Although presence of additional defenses (e.g., ASLR, stack-pointer protections [44,46]) will strengthen the impact of our defense, they are not necessary.

4.2. Key Observations

Our defense is based on two key observations regarding benign execution of programs:

O1 **Initialization of syscall arguments:** Most arguments to functions in general and system calls in particular are initialized in one or two basic blocks preceding the call/syscall instruction. More specifically, it is highly uncommon to find initialization of different arguments to be separated by *indirect branches*. However, in the case of CRAs, arguments are initialized in different gadgets that are *necessarily* separated by indirect branches (e.g., indirect jmp instruction in JOP, ret instruction in ROP).

O2 **Adherence to Conventions:** Programs are compiled using compilers that subscribe to pre-defined standards and ABIs. As such, code in programs adhere to calling conventions, especially the callee- and caller-saved register conventions as mandated by the ABI. An attack's gadget chains are under no obligation to, and often do not adhere to any such conventions.



```
;(EG1)function: foo
; type: entry point
push rbp          #rbp: 10
mov rbp, rsp      #rsp: 10
mov rax, rdi      #rax: 01
                  #rdi: 10
mov rdi, <addr> 1
mov rsi, $1024 2 #rsi: 01
call rax
pop  rbp
...
```

```
;(EG2)function: f2
; type: entry point
push rbp          #rbp: 10
mov rbp, rsp      #rsp: 10
mov rdx, $4 3     #rdx: 01
add rax, $132     #rax: 01
mov [rbp+8],rax ;vuln
pop rbp
ret
```

```
;(CP1)function: bar
; type: Call-preceded
push rbp
mov rbp, rsp
push rbx
mov rbx, rdi
call rbx        # fp()
pop rbx         # rbx: 01  P2a
4 mov rax, $10  # rax: 01
5 syscall  ──→  mprotect()
```

Figure 3: Working example demonstrating enforcement of Calling-Convention policy and System call policy. Gadgets EG1, EG2 and CP1 are used to demonstrate CFI-evading attack presented in CCFIR [22]. The arrows represent control flow.

```
camlBap_helpers__entry : call rdi # 11 insn
...
caml_curry2_1 : jmp rdx # 6 insn
...
camlBap_helpers__fun_323265 : call rdi # 11 insn
...
caml_curry2_1 : jmp rdx # 6 insn
...
camlBap_visitor__fun_8169 : call rdi # 11 insn
...
```

Figure 4: CRA-like pattern found in BAP, an application written in OCaml

```
.text : jmp qword ptr [rbx] # 5 insn
...
.text : jmp qword ptr [rbx] # 3 insn
...
.text : jmp qword ptr [rbp] # 5 insn
...
.text : jmp qword ptr [rbx-0x1] # 8 insn
...
.text : jmp qword ptr [rbp] # 3 insn
...
```

Figure 5: Repetitive jump pattern found in Pandoc, an application written in Haskell

4.3. System-Call Policy

Based on our observation **O1**, we define the system-call policy as follows:

P1: *Every argument to a system call must be populated at distance no greater than the threshold distance from the system call instruction.*

Here, distance is measured in terms of the number of indirect branch instructions between the system call and associated argument setting. In a nutshell, we monitor writes to system-call argument registers in the hardware, and associate a counter with each register. The counter represents *distance* in **P1**. Whenever a `write` operation occurs to a system-call argument register, the `distance` for the register is set to 0, and when an indirect branch instruction is encountered, the `distance` values of *all* argument registers are incremented. Finally, when a system call instruction is encountered, the `distance` values of each argument register are examined and validated against the policy. The corresponding algorithm is presented

```
_objc_msgSend : test rdi, rdi
_objc_msgSend : hint-not-taken jz 0x7fff736c4ee8
_objc_msgSend : test dil, 0x1
_objc_msgSend : hint-not-taken jnz 0x7fff736c4ef3
...
_objc_msgSend : jmp qword ptr [r11+0x8]
-[NSAppearance allowsVibrancy]: push rbp # method
-[NSAppearance allowsVibrancy]: mov rbp, rsp
...
```

Figure 6: The instruction trace of `_objc_msgSend` prior to the `allowsVibrancy` method call.

in Algorithm 1.

We examined multiple widely-used programs such as binutils, coreutils, gnome-web, etc. along with SPEC 2017 programs and empirically found the *threshold distance* to be 2 in most cases, which is unsurprising since system call invocations in benign code occur through system call wrappers or dispatcher functions in `libc`.

However, as a predominant property of CRAs, arguments are populated in multiple gadgets that are separated by one or more indirect branch instructions (indirect `call`/`jmp` or `ret`). In order for an attack to circumvent **P1**, it would need to populate all of the system-call arguments within the *threshold* distance, which is extremely hard (see Section 7.2).

Our system call policy **P1** has three key advantages. First, it captures the *essence* of code-reuse attacks, i.e., indirect branching, and is therefore extremely effective. Second, from a practical standpoint, tracking distance in the hardware is straightforward with *fixed storage overhead*. Finally, such a solution is *highly portable*. System V and MSVC ABIs–the two most popular ABIs are very similar in the way they utilize registers (see Figure 7) for argument passing. Therefore, our solution can port to other environments with little modification.

4.4. Calling-Convention Policy

Based on observation **O2**, we define a policy directed at adherence to calling convention. Benign programs adhere to an underlying ABI that mandates the calling convention that must be followed during function invocation. The calling convention dictates rules for saving and restoring registers,

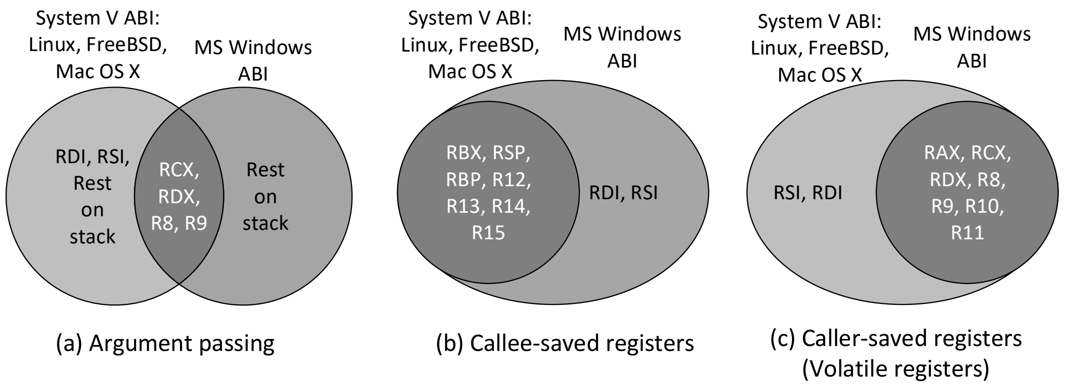

System V ABI: Linux, FreeBSD, Mac OS X
MS Windows ABI
RDI, RSI, Rest on stack
RCX, RDX, R8, R9
Rest on stack
(a) Argument passing
System V ABI: Linux, FreeBSD, Mac OS X
MS Windows ABI
RBX, RSP, RBP, R12, R13, R14, R15
RDI, RSI
(b) Callee-saved registers
System V ABI: Linux, FreeBSD, Mac OS X
MS Windows ABI
RSI, RDI
RAX, RCX, RDX, R8, R9, R10, R11
(c) Caller-saved registers (Volatile registers)

Figure 7: Convention for register use across function calls for System V and Microsoft ABIs.

passing arguments to caller and passing return value from a callee back to the caller. As a key insight, attacks rely on indirect `call` and `ret` instructions, but semantically, `call` and `ret` instructions indicate entry and exit from functions, and therefore the expectation is that the calling conventions are respected across function calls. However, code-reuse attacks do not follow these conventions. We derive the following rules from the ABI:

1. *Rule for callee-saved registers:* A callee-saved register must be saved by a callee before use. That is, from a hardware perspective *a callee-saved register must be read-from before being written-to*.
2. *Rule for caller-saved (or volatile) registers:* The ABI offers no guarantees that the contents of a volatile register or a caller-saved register will be preserved across function calls. As such, after a `ret` instruction in a callee function is encountered, *the caller function must not read from a volatile register (except while reading the return value) before first writing into it*. Conversely, *a callee function can not read from a non-argument volatile register before first writing into it.*
3. *Rule for arguments and return value: After a `ret` instruction, the caller function can only read from the return-value register (RAX) only if the callee function performed a write to the return-value register before the `ret` instruction. Similarly, after a `call` instruction, a callee function can not read-before-write from more number of argument registers than those that were written to by the caller function.* That is, if the caller function writes to the first two argument register, the callee function can not read from third or higher argument register before first writing into it.

While each of the rules can lead to a separate policy, not all rules can be effectively enforced in the hardware (see 4.4.1). In this paper, we focus on the callee-saved register rule. Specifically, we enforce the following policy:

P2: *Between successive `call` and `ret` instructions, callee-saved register must be read-from before being written-into.*

Given the substantial amount of overlap in calling convention policies for different environments (see Figure 7), our approach can be easily ported to most X86-64 environments (e.g., Mac OS X, UN*X, FreeBST, MS Windows). In a nutshell, we intercept instructions in hardware, and we check for read/write operations on registers between `call-ret` instruction pairs. A `call-ret` pair indicates an entry and exit from a function, so the *read-before-write* and *write-before-read* primitives will be enforced on callee-saved registers.

4.4.1. Policy Robustness and Compiler Optimizations In our experience, the callee-saved register policy is the most robust and conducive for hardware enforcement. Modern compilers employ aggressive optimizations that can relax some of the ABI convention rules. Since the compiler knows all the callees of a caller function during compile time, if the compiler can reason that a callee is not going to use a caller-saved volatile register, the compiler can optimize performance of the caller by not saving/restoring the caller-saved registers. In our experience, such optimizations for caller-saved registers are common and do not provide a robust basis for enforcement in the hardware. Whereas, in the case of callee-saved registers, it is not possible for a compiler to know all the callers of a function during compile time, therefore, callee-saved registers are always saved and restored within a function.

In the case of arguments and return value, return values can be ignored by the caller, and any write to return register may be interpreted as initialization of return value (even if the function does not return a value). In essence, without function signatures (which are not available in the hardware), reasoning about return values and arguments are non-trivial and incur performance overhead.

Therefore, in this paper, we focus on enforcement of the highly robust callee-saved register policy.

5. System Design

In this section, we provide more details of our tracking mechanisms.

5.1. System Call Defense

Policy Configuration We provide an *Argument-Specific policy* where we profile each system call to record the maximum depth for *each* argument, and generate a *System Call Table* that represents the highly granular and argument specific policy.

System Call Register Tracking The goal of system call register tracking is to track and record the initialization of various system call argument registers. The Algorithm 1 presents our technique for depth tracking. Particularly, we maintain a per-argument-register variable called *Depth* that records the distance from the system call instruction that a particular argument was set. When a write occurs on an argument register, *Depth* for that register is reset to 0 whereas when an indirect branch instruction is encountered, the depth of registers is increased to indicate the increase in distance from *syscall* instruction. Finally, before execution of a *syscall* instruction, the register depths are validated to ensure that they are in accordance with the policy.

For example, in Figure 3, writes to argument registers `rsi`, `rdi`, i.e., ① and ② happen in gadget **EG1** whereas write to register `rdx`, i.e., ③ occurs in **EG2**. Finally, the system call number is set in **CP1** ④ before the `syscall` instruction ⑤ is invoked. So, the depths for `rax` is 0 (i.e., the write happens 0 indirect branches away from the `syscall` instruction), `rdx` is 1, and `rdi` and `rsi` is 2. But as per the policy for `mprotect`, the expected depth for all arguments is 0 (see Figure 10). Therefore, an attack is inferred.

Algorithm 1: System call depth tracking

```
Data: Instruction $insn$
if $insn = \texttt{syscall}$ then
    $validatePolicy()$;
else
    if $insn$ writes to $reg \in SysCallArgumentRegister$ then
        $Depth[reg] \leftarrow 0$
    end
end
if $insn \in \{indirect\ jmp, indirect\ call, ret\}$ then
    $\forall reg \in SysCallArgumentRegister$
    $Depth[reg] \leftarrow Depth[reg] + 1$
end
```

Unequal Depths in Benign Code Although most arguments to system calls are set at depths 0 or 1, there are some cases where depths are higher.

Structure dereferencing: We performed a case study of the `read` system call that accepts 3 arguments through `rdi, rsi`, and `rdx` registers. The control flow leading up to the `__read` wrapper function in libc is as shown in Figure 8.

The `rsi` and `rdx` registers are written to when `buf` and `size` are set in `IO_file_read`. These are directly passed on to `__read`, and therefore depth is 1. Whereas, `fp→fileno` will cause a write to `rdi`, which makes the depth 0. More generally, when arguments leading up to a system call are initialized in different functions that are invoked via function pointers (i.e., indirect branching), such unequal depths are possible. In order to generate the policy, we profiled a

```
int IO_file_underflow(fp) {
 ....
 IO_file_read(fp, fp->IO_buf_base, fp->IO_buf_end - fp->
      IO_buf_base); /*---> This is an indirect call */
....
}

int IO_file_read(fp, buf, size) {
.....
__read(fp->fileno, buf, size);  /* fp->fileno is the file
      descriptor */
...
}
```

Figure 8: The case of `read` system call.

large corpus of real-world applications to determine maximum depths (i.e., threshold) for each argument to sensitive system calls in benign code (see Figure 10). Any execution at runtime that exceeds the threshold is a perceived attack.

Optional arguments: Consider the `futex` system call: `int futex(int uaddr, int futex_op, int val, struct timespec timeout, int uaddr2, int val3)`

Only the `uaddr`, `futex_op`, and `val` arguments are mandatory whereas `timeout`, `uaddr2`, `val3` arguments are optional and their presence depends on the value of `futex_op`. In such cases, the compiler will not populate optional arguments, and the corresponding argument register will contain a depth value corresponding to some past unrelated write to the corresponding register. Therefore, we only track mandatory arguments.

In case of optional arguments, the policy reserves a special bit value to indicate to the hardware that the depth of the register must be ignored during enforcement.

5.2. Calling-Convention Policy

The policy **P2** is extracted from the calling conventions presented in the X86-64 System V ABI document [38].

Step-by-Step Attack Inference for Running Example

Runtime Tracking We are interested in tracking the first access (write or read) that happens on a register within *each function frame*, i.e., between successive `call` and `ret` in the instruction stream. To this end, we intercept each instruction and record the read and write register operands of the instruction and accordingly generate shadow data. For each register, we maintain information per function frame to record read and write operations. Particularly, we are interested in identifying a *read-before-write* or a *write-before-read* behavior on a register.

Further, because register reads and writes are tracked per function frame, we maintain a shadow stack that stores individual frame-specific shadow data. The data for a frame is pushed and popped from the stack when `call` and `ret` instructions are encountered respectively.

Special cases: Instructions such as `xor rax, rax` read and write from the register at the same time. However, we are interested in reads that reflect the 'register saving' behavior

#	Insn	Inference	Bit vector after insn rax,rbx,rcx,rdx,rbp,rsi,rdi	Shadow Stack	Policy
1	push rbp	rbp: 10	00,00,00,00,10,00,00		P2: Pass
2	mov rbp, rsp		00,00,00,00,10,00,00		rsp: exempt
3	mov rax, rdi	rax: 01, rdi: 10	01,00,00,00,10,00,10		
4	mov rdi, <addr>		01,00,00,00,10,00,10		
5	mov rsi, $1024	rsi: 01	01,00,00,00,10,01,10		
6	call rax		00,00,00,00,00,00,00	01,00,00,00,10,01,10	
7	push rbp	rbp: 10	00,00,00,00,10,00,00	01,00,00,00,10,01,10	P2: Pass
8	mov rbp, rsp		00,00,00,00,10,00,00	01,00,00,00,10,01,10	rsp: exempt
9	mov rdx, $4	rdx: 01	00,00,00,01,10,00,00	01,00,00,00,10,01,10	
10	add rax, $132	rax: 01	01,00,00,01,10,00,00	01,00,00,00,10,01,10	
11	mov [rbp+8],rax			01,00,00,00,10,01,10	
12	pop rbp			01,00,00,00,10,01,10	
13	ret		01,00,00,00,10,01,10		
14	pop rbx	rbx: 01	01,01,00,00,10,01,10		**P2: Fail**

Table 1: Attack trace in Intel syntax for running example in Figure 3 with register read/write tracking and calling-convention policy. The value '10' represents read-before-write and '01' represents write-before-read.

within a function frame, and as such, we associate a *write-before-read* primitive with such instructions. Additionally, we treat `rsp` register different from other registers. Although `rsp` is a callee-saved register, a *write-before-read* can occur when a program is compiled without frame pointer `rbp`, and stack space is allocated. Therefore, we exempt stack pointer from the policy.

Policy Enforcement When an instruction is encountered, the register reads and writes are evaluated to test for compliance with calling-convention policies in Section 4.4.

An instruction-by-instruction inference and stack contents for the running example in Figure 3 is presented in Table 1. The read/write inferences are made after each instruction, and finally, when the `pop rbx` instruction is encountered in gadget **CP1**, it is inferred as a *write-before-read* for a callee-saved register `rbx`, which triggers a policy violation per **P2**.

5.3. Multi-Threading and Multi-Process Support

Although we do not explore multi-threading and multi-process support in this work, we believe our solution can be easily extended to support multiple threads and processes including. Specifically, the thread control block (TCB) and process control block (PCB) can be modified to save the bit vector and shadow stack as a part of the context information, so that the enforcement states can be saved and restored across multiple threads and/or processes. Appropriate changes to runtime and OS will be necessary.

5.4. Handling Exceptional Flows and Hand-Written Assembly

During exceptional flows like `setjmp/longjmp`, a large set of registers are read from (during `setjmp`) and are restored (during `longjmp`) without regard to conventional norms. A similar case manifests in the case of hand-written assembly. In totality, such code instances are extremely small and typically well defined (e.g., low-level kernel routines). We propose to profile them before-hand and generate a signature that is made available to the hardware for exclusion.

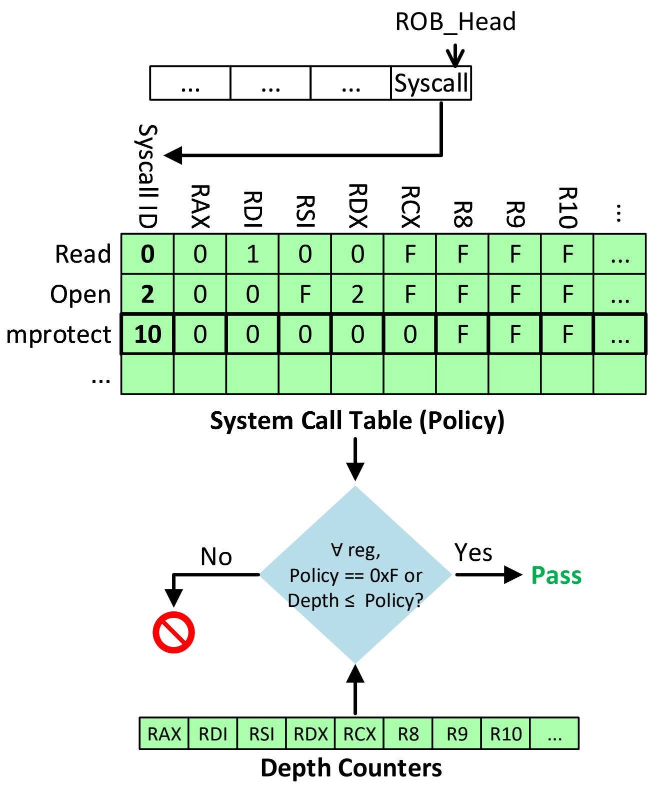
6. Hardware and Microarchitectural Support and Considerations

In this section, we describe simple microarchitectural changes required to implement our approach. As we demonstrate, the amount of hardware needed is modest.

6.1. Tracking Argument Depths for System Calls

For the system call tracking approach, the complexity depends on the number of system calls that are tracked by the detection system. As shown in Figure 9, the key structure to support our syscall tracking is a table that is maintained at the commit stage of the pipeline, we call it System Call Table (SCT). The number of rows in SCT equals to the number of supported system calls, and the number of columns equals to the number of registers used as system call arguments, plus a column to store a system call number to use as a search tag for the system call. Previous research published in security community established most security-critical system calls to be relatively few [17, 24, 53] (specifically: `execve, write, mprotect, munmap, clone, fork, open, close, exit_group, read`), so that most attacks can be successfully prevented if only a few system calls are tracked. SCT forms the policy that provides expected argument depth values (i.e., threshold) for each monitored system call.

While the size of SCT can be configured to cover different



	Syscall ID	RAX	RDI	RSI	RDX	RCX	R8	R9	R10	...
Read	0	0	1	0	0	F	F	F	F	...
Open	2	0	0	F	2	F	F	F	F	...
mprotect	10	0	0	0	0	0	F	F	F	...
...										

Figure 9: Syscall Depth Policy Enforcement

number of system calls, for our calculations we assume that a 16-entry SCT is used. Inside each entry, we store a system call number and system call arguments. In x86-64 architecture, the arguments are stored in registers rdi, rsi, rdx, r10, r8, and r9, so we assume six register depths are stored for each system call in that order. If each depth value requires 4 bits to express the depth, plus 9 bits are needed to record the system call number (assuming 512 system calls), then each SCT entry will require 33 bits of storage (which can be rounded to five bytes resulting in 80 bytes of storage for a 16-entry SCT). SCT can be organized as a fully-associative or a set-associative structure. For the small size of 16 entries we use a fully associative search on the system call number. SCT is loaded only once for each execution environment (for example, when OS boots) and it provides reference information against which the register depth counters collected at runtime are compared to make security decisions.

At runtime, we also need to track the depth of every register used as a system call argument. Our tracking captures the depth of each register between consecutive system calls. Each ISA register is associated with its own *depth counter*. There are 4-bit long saturating counters. After a system call instruction is committed, all depth counters are reset to zero. Whenever a write to a register occurs, its depth counter is also set to zero. Whenever an indirect jump, an indirect call or a return instruction commits, the depth counters of all registers are incremented by one. At the commit time of the next system call (the one that is being tracked), we read depth counters corresponding to the system call arguments from the depth counters and compare them from the information for this syscall in SCT. Since all system calls use standard conventions for register usage, the same ISA registers are always checked. If a particular register is not used for a given system call (because the number of arguments is smaller or the argument is optional), this is indicated by a reserved bit-sequence for that register in SCT (4 bit value 1111 or 0xF). A variety of policies can be implemented based on the values of the counters, ranging from simple to more complicated ones.

Note that these additional hardware resources are manipulated at the commit stage of the instruction pipeline. Since all accesses occur at the commit stage, these accesses are off the critical schedule-to-execute timing path. The table can be configured and sized to be accessed within a single cycle. Even if an additional cycle or two are needed, the commit stage can be pipelined into several stages without impacting the number of instructions committed per cycle, as this does not lengthen the critical fetch-to-execute loop and does not impact the branch misprediction penalty [8]. To reduce the size of the system call table, one can track only the most security-critical system calls. This will simplify the logic, but still significantly reduce the attack surface.

The above scheme only tracks and detects CRAs based on non-speculative gadgets. If speculative gadgets need to be considered to protect from some forms of transient execution attacks, our support can be easily extended by moving the monitoring logic to the front-end of the pipeline (decode stage) and making appropriate adjustments to the depth counters on branch misspeculations (similar, in principle, to how a rename table and a free list of physical registers is recovered on branch misspeculation). Note that SCT does not need to be adjusted, since this is not a writable structure during normal execution.

6.2. On-Chip Storage Overhead

For **P1** *system-call specific* policy, the overhead scales with the number of system calls monitored. As described above, with 80 bytes of storage we can implement support for 16 most critical system calls. As the number of system calls increases, so does the storage requirement for SCT.

6.3. Software Configuration

We allow software configuration of SCT. For example, SCT contents can be set differently for various operating systems at system boot time. Furthermore, a more fine-grain reconfiguration using privileged system call interface is also possible. This is no different than any other system with configurable hardware parameters.

7. Evaluation

In this section, we present the performance, complexity, and security evaluation.

7.1. Performance Analysis

The performance overhead of tracking ABI conventions stems from misses from the hardware shadow stack. To estimate the additional number of cycles incurred by such accesses, we simulated our system using Pin binary instrumentation tool [27]. We ran each SPEC 2017 benchmark through the Pin tool for 1 billion instructions. For each benchmark, we simulated with hardware stack of 2, 4, 8, and 16 entries, and we kept the cache configuration consistent. We used a `64kB` L1 data and instruction cache, a `512kB` L2 cache, and a `2MB` L3 cache. The assumed access latencies for different memory levels were: 1 cycle for L1 cache, 20 cycles for L2 cache, 35 cycles for L3 cache, and 200 cycles for DRAM. To calculate the total cycle penalty, we observed how often the ABI enforcement mechanism misses into the hardware stack, and from which level of memory the misses were serviced.

The Table 2 shows that the overhead due to hardware shadow stack misses had negligible impact on the average memory access time (AMAT) of the system. The most significant difference in performance was seen for the `520.omnetpp_r` benchmark for a shadow stack size of 2 entries which resulted in the AMAT increasing by 0.35% when compared to the baseline. These results can be attributed to the low recursion depth as a result of which there are fewer entries in the stack. The fewer entries result in a larger number of stack accesses serviced by the hardware shadow stack and L1 cache. As one would expect, a hardware stack size of 16 entries resulted in the best AMAT, with the worst performing benchmark, `520.omnetpp_r` facing an AMAT increase of just 6.2e-05% when compared to the baseline.

As the ABI compliance check utilizes the stack to store the state of register accesses across function calls, benchmarks exhibiting deep non-tail-recursive calls result in greater memory usage. These exceptional benchmarks however, only manage to cause insignificant losses in cache performance.

Additionally, the memory overhead incurred by use of shadow stack is presented in Figure 13. For most programs in Spec 2017, the burden was less than 1KB.

7.2. Security Analysis

We analyze the impact of our defense on the overall security of the system. Particularly, we examine the reduction in attack surface due to incorporation of *System Call Depth* (i.e., **P1**) and *ABI Compliance* (i.e., **P2**) policies. We analyze the feasibility of execution of system calls in a code-reuse paradigm. To this end, we examine the system calls in Linux and evaluate how much harder it will be for an attacker to accomplish an attack—i.e., execute the system calls through code-reuse attacks—in the presence of our defenses.

Methodology For a given program, we first compute the total possible gadget chains in the program's address space that can be used invoke each system call. We follow these steps:

AMAT Increase Due to Shadow Stack

Program	Number of Shadow Stack Entries			
	2	4	8	16
spec17				
`blender`	0.000335	0	0	0
`bwaves`	3e-05	0	0	0
`cactusBSSN`	0.000107	4.5e-05	1.3e-05	4e-06
`cam4`	0	0	0	0
`cpugcc`	0.00588	0.000278	0	0
`cpuxalan`	0.005586	0.00061	0.000114	0
`deepsjeng`	0	0	0	0
`exchange2`	0	0	0	0
`fotonik3d`	0	0	0	0
`imagick`	0	0	0	0
`lbm`	1e-06	1e-06	0	0
`leela`	2.2e-05	0	0	0
`mcf`	2.5e-05	2.2e-05	0	0
`nab`	2.5e-05	1e-06	0	0
`namd`	0	0	0	0
`omnetpp`	0.35306	0.24711	0.005459	6.2e-05
`parest`	6e-05	6e-06	4e-06	0
`perlbench`	0.002193	0.00026	0	0
`povray`	0.028826	1e-06	0	0
`roms`	2e-06	0	0	0
`wrf`	0	0	0	0
`x264`	2.4e-05	2.2e-05	0	0
`xz`	0.002067	0.000522	0	0

Table 2: Performance impact of the ABI Compliance Check on AMAT

- We start with a set of all the gadgets in all the libraries in a process' memory ($\mathscr{G}$). This includes all the gadgets in all the libraries in the process memory plus the program executable.
- We identify the set of syscall gadgets ($G_S \subset \mathscr{G}$) that can be used to invoke a system call, i.e., the last instruction in the gadget is the `syscall` instruction. For any successful system call invocation, $\exists g_s \in G_S$ where g_s is the last gadget in the gadget chain.
- We then identify a set of gadgets ($G_{RAX} \subset \mathscr{G}$) that must either load an arbitrary value into the `r/eax` register, or load a fixed value corresponding to a valid system call number. If a fixed value is loaded, then the gadget is only usable to invoke the system call whose number is loaded into `r/eax`.
- Similarly, we assemble argument-register sets of gadgets ($G_{RDI}, G_{RSI}, G_{RDX}, G_{R10}, G_{R8}, G_{R9} \subset \mathscr{G}$) that can load a value into the system call argument registers. That is, rdi, rsi, rdx, r10, r8 and r9 registers, or rax, rbx, rcx, rsi, rdi and rbp for legacy X86 system calls that use `int 0x80` as their system call instruction. Finally, we identify a chain of *smallest number of gadgets* that can be used to initialize system call arguments depending on how many arguments the given system call accepts.

Target Programs: The SPEC benchmark is not best suited for security evaluation, as such we picked real-world programs mysql and Firefox, wherein we tested *all* loaded libraries along with Firefox and mysql executables. Our analysis recreates the exploitation environment an attacker would encounter while exploiting mysql or Firefox. Additionally, our solution has

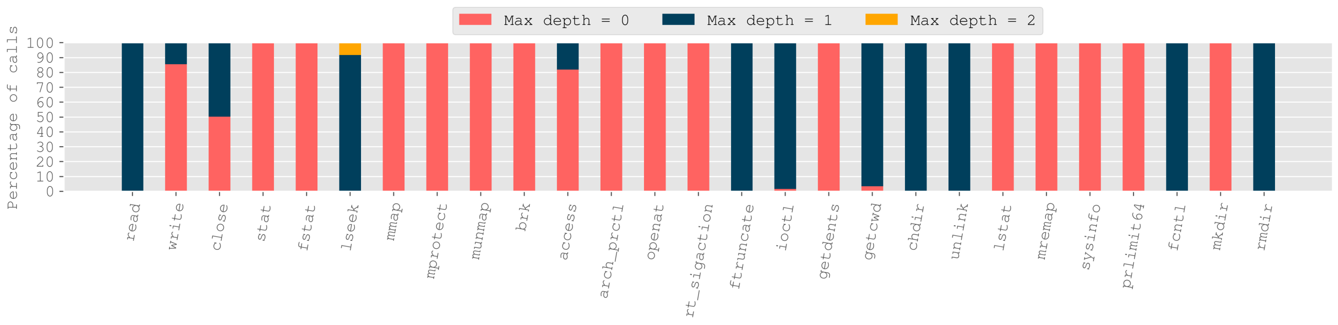


Figure 10: Max System Call Argument Depth for Linux System Calls.

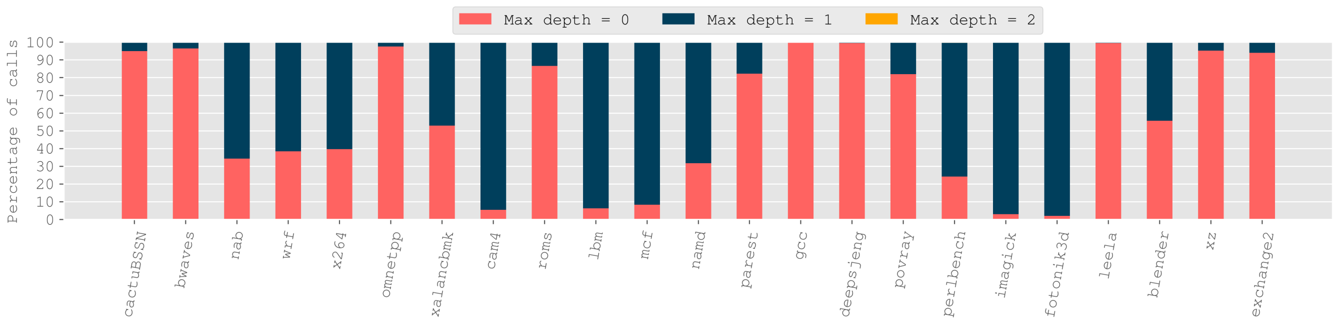


Figure 11: Max System Call Argument Depth for SPEC 2017 benchmark programs.

been tested on libraries used by perl and python.

Attack surface reduction We followed the steps above to compute the individual gadget sets for all the libraries used by `mysql` database program and computed the smallest possible gadget chains with and without the presence of our defenses. Next, we computed the chain lengths for execution of system calls with 0 through 6 arguments using ROP and JOP. Because system call depths are typically no more than 3 (see Figures 10 and 11), we restricted to depth of 3. Our findings are presented in the logarithmic graph in Figure 12. Without **P1, P2** defenses, the number of gadget chains possible are 2.56×10^{48} and 1.03×10^{17} for ROP and JOP attacks respectively. These are nothing but the product of cardinalities of $G_{RDI}, G_{RSI}, G_{RDX}, G_{R10}, G_{R8}, G_{R9}$.

Key finding: The number of possible gadget chains with **P1 + P2** for both ROP and JOP at depth 0 drops to 1, which corresponds to the system call wrapper function in libc. Given that most system calls set arguments at depth 0, this finding suggests that *our solution entirely prevents execution of most system calls using CRAs*.

Additionally, we examined the impact of **P2** on reduction of number of usable gadgets that write to callee-saved registers in the Firefox browser. Our findings are tabulated in Table 3. We see that just by application of **P2** we can eliminate 88.7% and 99.3% of gadget chains that operate on callee-saved `rbx` and **rbp** registers respectively. Note that although `rbx` and `rbp` registers are not directly involved in argument passing to a system call, they are often read-from or written-to as a side-effect in gadgets that are useful for argument initialization. Enforcement of additional policies is only expected to further reduce attack surface. Further, it should be noted that the mentioned gadgets can be used for system calls that use `int 0x80` as their system call instructions.

Comparison against Intel's CET and Hurdle Intel CET [26] focuses on source-target mappings for correct control flow, whereas our solution relies on conventions, which is robust and fundamentally different (yet orthogonal) to CET's approach. Unlike Hurdle [20], *our approach can defend against all types of CRAs (return-, jump- and call-oriented programming) without any modifications to the binary*, thereby providing full backward compatibility. Also, Hurdle is unable to protect against data-only code-reuse attacks whereas our solution through enforcement of P2 can stop such attacks.

Firefox	**RBX (no defense)**	**RBX (with P2)**	**RBP (no defense)**	**RBP (with P2)**
libc	1031	111	616	2
libdl	45	12	21	0
libgcc_s	96	0	75	0
libstdc++	674	139	419	5
libm	565	7	324	2
libpthread	96	15	83	0
Total	**2507**	**284**	**1538**	**9**

Table 3: Gadget reduction for callee-saved registers with P2 enforcement.

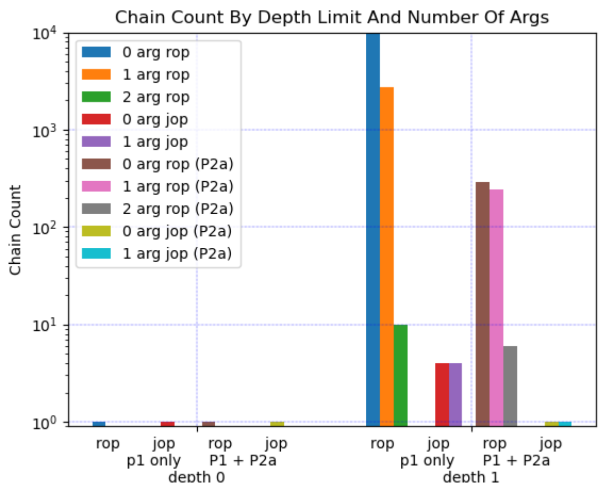


Figure 12: Impact of defense on attack surface reduction with P1 only, and with P1 + P2.

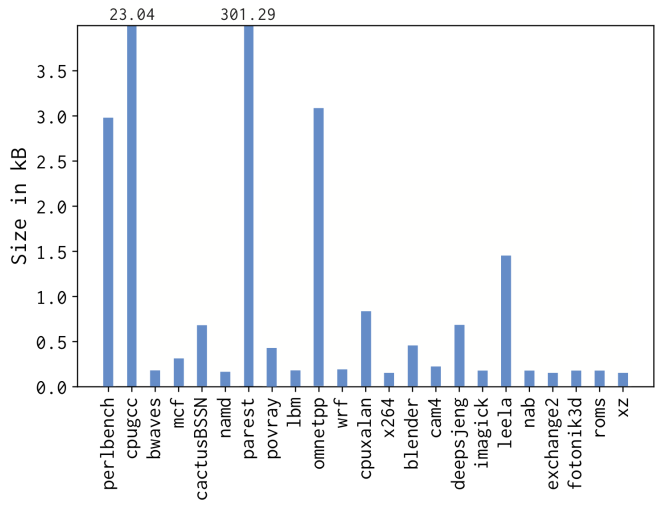


Figure 13: Memory Requirements of the Shadow Stack

8. Related Work

Previous signature-based defenses that target ROP attacks, [17, 19, 33, 43], look for inherent ROP behaviors that deviate from benign execution. Since JOP attacks do not rely on the return instructions, such attacks are capable of bypassing ROP defenses [7, 16] and require alternative defenses.

JOP-alarm [54] introduced the concept of a score value to detect a potentially malicious behavior representing a JOP attack. The score value is dynamically adjusted based on the gadget lengths and indirect jump distances. While being a promising concept, the score-based approach does not completely eliminate false positives. In contrast, our technique avoids false positive alarms by design. Tiny Jump-oriented programming (Tiny JOP) [47] performs JOP with very few gadgets, bypassing gadget thresholds in selected defenses. However, Tiny JOP is very specific to 32-bit x86 systems and is only capable of performing system calls that have the system call number encoded in the final gadget. Block-oriented programming attacks [28] construct the attacks using basic blocks as gadgets but violate **P1** and are therefore vulnerable to our approach.

Control-Flow Integrity (CFI) [1, 11, 45] is another way to prevent malicious control-flow changes by ensuring that the program execution adheres to its control flow graph (CFG). CFGs have been used to enforce CFI in various solutions [4, 18, 36, 51, 53]. In theory, CFI offers perfect control-flow protection, however in practice, CFI implementations are known to be inadequate [34]. The limitations of CFI have been documented in [14, 35]. In addition, unintended instructions would not be included during static analysis, even though most gadgets are unintentional [30]. Other defenses also require recompilation [5, 30, 42], which increases the code size and makes protecting legacy binaries more difficult. Memory bounds checking [21, 40, 41] is another comprehensive and fairly complex technique for protecting systems from buffer overflows.

Prior works also addressed system call checking for security purposes. Seccomp (Secure Computing) module performs system call checking in Linux implementations. The goal is to limit the range of system calls and arguments that a given process can invoke during execution. Software checks of Seccomp involve significant performance overhead, so hardware-supported checking acceleration has been recently proposed to address performance issues [50]. This type of system call checking is orthogonal to our approach.

9. Concluding Remarks

We demonstrated, through concrete examples, that signature-based detection schemes are not effective against code reuse attacks, because benign programs sometimes exhibit gadget-like behavior which is indistinguishable from attacks. Instead of tracking attack signatures, we showed that a more sound and effective detection approach is to track deviations from established execution conventions that govern the execution of regular programs. Specifically, we considered two forms of such tracking: system call argument depth, and compliance with ABI calling conventions. We showed that the attack surface can be significantly reduced with modest modest performance overhead and about 80 bytes of on-chip storage if 16 most security-critical system calls are tracked.

Acknowledgement

We would like to thank anonymous reviewers for their valuable feedback. We would also like to thank Ms. Atsuko Shimizu for insightful discussions and contributions. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929, DARPA award #81192, and NSF Award #2053391. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," *CCS '05 Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, 2005.

[2] Apple, "objc_msgsend," https://developer.apple.com/documentation/objectivec/1456712-objc_msgsend.

[3] ——, "Source browser," https://opensource.apple.com/source/objc4/objc4-532.2/.

[4] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1295–1308, 2006.

[5] W. Arthur, B. Mehne, R. Das, and T. Austin, "Getting in control of your control flow with control-data isolation," *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, 2015.

[6] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel {SGX}," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1213–1227.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," *ASIACCS '11 Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, 2011.

[8] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002, pp. 299–310.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[10] 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*. ACM, 2008, pp. 27–38.

[11] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, 2017.

[12] "Capstone the ultimate disassembler," http://www.capstone-engine.org.

[13] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[14] ——, "Control-flow bending: On the effectiveness of control-flow integrity," *Proceeding SEC'15 Proceedings of the 24th USENIX Conference on Security Symposium*, pp. 161–176, 2015.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[16] ——, "Return-oriented programming without returns," *CCS '10 Proceedings of the 17th ACM conference on Computer and communications security*, pp. 559–572, 2010.

[17] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. D. Huijie, "Ropecker: A generic and practical approach for defending against rop attack," *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, pp. 1–14, 2014.

[18] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," *CODASPY '16 Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 38–49, 2016.

[19] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: a detection tool to defend against return-oriented programming attacks," *ASIACCS '11 Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 40–51, 2011.

[20] C. DeLozier, K. Lakshminarayanan, G. Pokam, and J. Devietti, "Hurdle: Securing jump instructions against code reuse attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[21] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 103–114, 2008.

[22] E. Göktaş, E. Anthanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)*, 2014.

[23] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 417–432.

[24] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017.

[25] Y. Guo, L. Chen, and G. Shi, "Function-oriented programming: A new class of code reuse attack in c applications," in *2018 IEEE Conference on Communications and Network Security (CNS)*, May 2018, pp. 1–9.

[26] intel, "Control-flow enforcement technology preview," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[27] Intel, "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[28] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[29] ——, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1868–1882. [Online]. Available: https://doi.org/10.1145/3243734.3243739

[30] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, 2012.

[31] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium*, pp. 258–269, 2013.

[32] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. A. Ghazaleh, "Signature-based protection from code reuse attacks," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 533–546, 2015.

[33] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on arm mobile devices," *HASP '15 Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, 2015.

[34] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[35] ——, "Finding cracks in shields: On the security of control flow integrity mechanisms," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1821–1835.

[36] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, 2017.

[37] J. MacFarlane, "Pandoc a universal document converter," 2006. [Online]. Available: https://pandoc.org/

[38] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System v application binary interface amd64 architecture processor supplement," https://uclibc.org/docs/psABI-x86_64.pdf, Nov 2014.

[39] Microsoft, "Data execution prevention," https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx, 2003.

[40] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[41] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: An empirical study of intel mpx and software-based bounds checking approaches," *arXiv preprint arXiv:1702.00719*, 2017.

[42] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," *ACSAC '10 Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 49–58, 2010.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," *SEC'13 Proceedings of the 22nd USENIX conference on Security*, pp. 447–462, 2013.

[44] A. Prakash and H. Yin, "Defeating rop through denial of stack pivot," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 111–120.

[45] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 311–322.

[46] A. Quach, M. Cole, and A. Prakash, "Supplementing modern software defenses with stack-pointer sanity," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 116–127.

[47] A. Sadeghi, F. Aminmansour, and H. R. Shahriari, "Tiny jump-oriented programming attack (a class of code reuse attacks)," *Information Security and Cryptology (ISCISC), 2015 12th International Iranian Society of Cryptology Conference on*, 2015.

[48] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 745–762.

[49] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," *CCS '07 Proceedings of the 14th ACM conference on Computer and Communications Security*, pp. 552–561, 2007.

[50] D. Skarlatos, Q. Chen, J. Chen, T. Xu, and J. Torrellas, "Draco: Architectural and operating system support for system call security," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 42–57.

[51] D. Sullivan, O. Arias, L. Davi, P. L. A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity," *DAC '16 Proceedings of the 53rd Annual Design Automation Conference*, no. 163, 2016.

[52] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[53] V. van der Veen, D. Andriesse, E. Gökta, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," *CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[54] F. Yao, J. Chen, and G. Venkataramani, "Jop-alarm: Detecting jump-oriented programming-based anomalies in applications," *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[55] D. Zheng, "A look under the hood of objc_msgsend()," http://blog.zhengdong.me/2013/07/18/a-look-under-the-hood-of-objc-msgsend/, 2013.