SCRAP: Architecture for Signature-Based Protection from Code Reuse Attacks

Mehmet Kayaalp, Timothy Schmitt, Junaid Nomani,
Dmitry Ponomarev and Nael Abu-Ghazaleh
Computer Science Dept.,
Binghamton University
{mkayaalp,jnomani1,tschmitt1,dima,nael}@cs.binghamton.edu

Abstract

Code Reuse Attacks (CRAs) recently emerged as a new class of security exploits. CRAs construct malicious programs out of small fragments (gadgets) of existing code, thus eliminating the need for code injection. Existing defenses against CRAs often incur large performance overheads or require extensive binary rewriting and other changes to the system software. In this paper, we examine a signature-based detection of CRAs, where the attack is detected by observing the behavior of programs and detecting the gadget execution patterns. We first demonstrate that naive signature-based defenses can be defeated by introducing special "delay gadgets" as part of the attack. We then show how a software-configurable signature-based approach can be designed to defend against such stealth CRAs, including the attacks that manage to use longer-length gadgets. The proposed defense (called SCRAP) can be implemented entirely in hardware using simple logic at the commit stage of the pipeline. SCRAP is realized with minimal performance cost, no changes to the software layers and no implications on binary compatibility. Finally, we show that SCRAP generates no false alarms on a wide range of applications.

1. Introduction

Exploits targeting software vulnerabilities remain one of the primary security threats to computer systems, with costs estimated in the 100s of billions of dollars [5].The NIST national vulnerability database includes tens of thousands of vulnerabilities, with an average reporting rate of 10 new vulnerabilities per day [40]. Thus, it is critical to build systems that make exploits difficult to launch and that detect and limit their effect quickly.

Most current attacks start by exploiting a buffer overflow vulnerability. Despite significant efforts in devising solutions that prevent buffer overflows [18, 22, 23, 52], they remain prevalent. Early code injection attacks overwrote the buffer with the malicious code on the stack and simultaneously overwrote the return address to point at the start of the exploit code [3,43]. A number of software and hardware approaches to protect against such attacks were devised [6, 15, 17, 44]. These efforts have culminated in the recent deployment of hardware memory protection mechanisms that do not allow a memory page to be both writable and executable at the same time (the so called $W \oplus X$ protection). These hardware extensions are supported by both AMD and Intel processors and deployed in both Linux and Windows operating systems [4, 54].

1.1. Code Reuse Attacks: Bypassing $W \oplus X$

In response to these defenses new Code Reuse Attacks (CRAs) emerged that construct a malicious program by stitching together carefully selected fragments of the existing library code; these snippets are called *gadgets* [47]. One example of a CRA is the return-oriented programming (ROP) attack, where each gadget ends with a return instruction to trigger the execution of the next gadget pointed to by the next return address on the stack. All the attacker has to do is to inject a proper sequence of return addresses onto the stack to point to the needed gadgets. ROP was shown to be Turing-complete on a variety of platforms [9, 12, 19, 27, 34]. Automated tools have been developed that allow unsophisticated attackers to construct arbitrary malicious programs using ROP [24, 25, 31, 46].

Several defense mechanisms against ROP have been recently proposed [13, 20, 33, 35, 49, 58]. Perhaps the simplest of these solutions are the ones that utilize a shadow call/return stack, where the return instructions are matched against the corresponding calls using protected memory space [35, 49, 58]. We assume that such an enforcement of call-return pairs is already in place and therefore simple ROP-based attacks are defeated.

Unfortunately, a new form of CRA was developed that does not rely on return instructions [8, 11, 14]. In this jump-oriented programming (JOP) model, the attacker chains the gadgets using a sequence of indirect jump instructions, rather than return instructions. A special dispatcher gadget is used to orchestrate the control flow among the gadgets. A high level example of the JOP attack model is shown in Figure 1. This diagram shows how the attack will jump from the dispatcher gadget to functional gadgets which will then return the control back to the dispatcher gadget. The jump locations change based on the addresses popped off the stack by the dispatcher gadget, and will ultimately result in the execution of a system call.

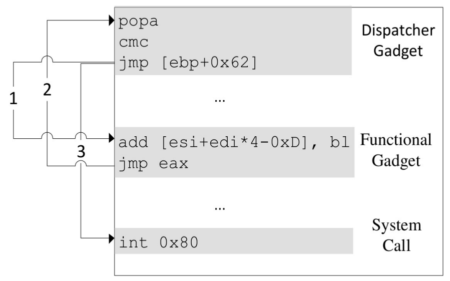

Figure 1. Example of a simple JOP attack

1.2. Proposed Solution: Signature-based CRA Detection

Although it may appear that CRAs are a narrow form of attack, they represent a wide-open vulnerability that is increasingly used to exploit common buffer overflows. For example, many recent approaches to jailbreak and software unlock smart phones are CRA-based [50]. Thus, it is critical to develop solutions that protect against this major vulnerability, preferably in a way that protects legacy binaries.

In this paper, we propose Signature-based CRA Protection (SCRAP): a simple and low-overhead hardware scheme to protect against JOP attacks that are based on the dynamic detection of attack signatures, or the patterns of executed instructions that are indicative of the JOP attack. SCRAP works because the attack patterns are significantly different from those of the regular programs. Previous work [13] investigated this type of defense for ROP attacks and showed that it has promise.They implemented a defense mechanism called DROP in software using Valgrind tool to detect the ROP pattern. Because it is implemented in software, DROP incurs over 5X performance loss on the average across simulated workloads, mainly due to the overhead of Valgrind.

Starting from DROP, we made several observations about existing signature-based detection that motivated this work. First, the ideas of signature-based detection can be extended to protect against the JOP attacks if one uses the indirect jumps as the gadget boundaries. Second, the high performance overhead of DROP (appropriately adapted to protect against JOP attacks) can be avoided by implementing the checking logic in hardware, placing this hardware off-the-critical path in the commit stage of the pipeline, and performing simple checks during instruction commitment. If successful, this approach can provide protection with much lower overhead and complexity compared to the previous solutions and can naturally protect the existing binaries. Third, and most important, the naive implementations of the signature-based detection along the lines of DROP can be bypassed because of the strong assumptions it makes about usable gadget lengths. For example, we demonstrate an attack that uses a delaying gadget through a function call in the middle of the attack with the only purpose to distort the attack signatures expected by a DROP-like signature-based defense. Finally, the thresholds on the length of gadgets assumed by DROP are not absolute: although difficult, it is possible to find longer gadgets and integrate them into an attack, avoiding detection. In this paper, we present a complete working example of such a stealth JOP attacks integrating delay gadgets, and using gadgets longer than the DROP thresholds.

Motivated by these observations, we propose an attack signature detection logic that protects against such stealth JOP attacks by filtering out the spurious function calls in the middle of the attack from the attack signature. We develop a language for the possible attack sequences and derive from it a state machine implementation of the detection logic. We show that the proposed mechanism generates no false alarms in any of the regular workloads that we considered and successfully detects CRAs, even when delay gadgets are used, for a large number of shellcodes. Finally, we extend the detectors to tolerate infrequent use of longer gadgets.

SCRAP has the following key characteristics:

- It successfully detects all JOP attacks that we were able to generate, while resulting in zero false alarms across regular code base.
- It incurs minimal performance cost (less than 2%) and only requires simple hardware at the commit stage of the pipeline. There is also no impact on the processor cycle time.
- It does not require complex binary rewriting, binary annotation, or construction of a full control flow graph of a program. It also does not require compiler or ISA support and can be used to protect legacy binaries.
- With a simple hardware support, it performs checks for unintended jumps (in variable instruction-length architectures, such as x86) thus closing the potential security vulnerability of purely software-based solutions.

2. CRA Mechanics and Example

In this section, we overview a fully functional example of a JOP attack. We follow by discussing how variable length ISAs such as x86 and x86-64 significantly increase the number of gadgets available for attacks.

2.1. Functional JOP Attack Example

Figure 2 shows an example of the malicious shell code to be executed by the attacker. The purpose of this simple code is to execute a system call that starts a new shell. As we discussed previously, since the attacker can no longer directly inject this code and execute it, this code has to be assembled from the existing library code. For this example, we use the standard C library (`libc 2.11.3`) as the code base for the gadget composition. Table 1 shows the gadgets that we found in libc to carry out the functionality of the attack from Figure 2. Finally, we show the dynamic sequence of the discovered gadgets to execute this attack and explain the functionality and purpose of each dynamic gadget invocation.

	Gadget	Gadget Function

g0	`popa`
	`cmc`
	`jmp [ebp+0x62]`
		Dispatcher

g1	`add [esi+edi*4-0xD], bl`
	`jmp eax`
		Null-Writer

g2	`int 0x80`	System Call

Table 1. Gadgets Used in Example Attack

In order to launch a shell using the gadgets in Table 1,

```
; Load 0x0000000b (the syscall number) to register eax
xor eax, eax
mov al, 0x0b

; Point ecx and edx to a null word, 0x00000000
mov ecx, 0xf7fc5fe3
mov edx, 0xf7fc5fe3

; Point eax to the string "/bin/sh"
mov ebx, 0xffffd6e9

; Make a system call to execve
int 0x80;
```

Figure 2. Example Shellcode in Assembly

this type of attack has to accomplish two things: the correct parameters for a system call must be placed in the argument registers and a system call must be made. To launch a shell, our example attack makes a system call to `execve`. When the system call is made, registers `ecx` and `edx` must point to a null word, `0x00000000`, and `ebx` must point to the string `"/bin/sh"`. Both null words and the string `"/bin/sh"` can be found in memory; we can place their addresses onto the stack and let the JOP attack pop them into the appropriate registers. The remaining step in the attack is to initialize the value of the `eax` register.

When the system call is made, `eax` must contain `0x0000000b`, indicating a call to `execve`. However, a JOP attack typically depends on exploiting a buffer overflow; these attacks typically rely on a buffer overflow which is exploited by the attacker to place data on the stack. The buffer is typically a string buffer, so a 0x00 byte causes the system to terminate reading the string; the attacker cannot use null values in the initial overflow. If the attack needs any null values, such as those in the word `0x0000000b`, the attack must generate them itself.

We make use of a *null-writer* gadget to create null values on the stack that will eventually be popped into `eax`. Our null-writer is constructed with an `add` instruction, adding the byte held in `bl` to the byte on the stack pointed to by `esi+edi*4-0xD`. If we place bytes holding `0xff` on the stack as part of the initial overflow attack and ensure that `bl` contains `0x01`, we can add `0x01` to `0xff` on the stack, overflowing to a `0x00`. Using this method, our attack creates the word `0x0000000b` on the stack where it can be popped into `eax` as the final step before the system call gadget is used.

In the remainder of this section we show how the attack executes using the gadgets described in Table 1. We assume the attacker has exploited a buffer overflow to place data on the stack and redirect control flow to the dispatcher gadget (g0). From the dispatcher gadget, the attack proceeds to execute the null-writer gadget (g1), then g0, g1, g0, g1, g0, and finally, the system call (g2). Below, each step starts with the gadget number followed by an explanation of how it advances the attack.

Step 1 - g0 The dispatcher gadget initiates the attack with a `popa` instruction. This instruction populates the registers with useful values the attacker has placed on the stack. The second instruction, `cmc`, has no meaningful effect on this attack. After initializing the registers with values necessary for an attack, the dispatcher jumps to the null-writer gadget.

Step 2 - g1 The null-writer gadget adds the byte held in `bl` to the byte that `esi+edi*4-0xD` points to. In Step 1, the dispatcher gadget populated the registers so that `bl` contains `0x01` and `esi+edi*4-0xD` points to the value `0xff` in the future value of `eax` on the stack.

Step 3 - g0 Populate the registers with the values necessary to perform the null-writer a second time.

Step 4 - g1 Write `0x00` to a second byte in the future value of `eax`.

Step 5 - g0 Populate the registers with values for a third and final execution of the null-writer.

Step 6 - g1 Write the final null value onto the stack where `eax` is popped from.

Step 7 - g0 Populate the registers with the appropriate values for a system call. The value that is popped from the stack to `eax` is `0x0000000b`.

Step 8 - g2 Make a system call to `execve()`, launching a new shell.

While the above example represents a straightforward attack (we chose this for the ease of demonstration), more sophisticated shellcodes, including the ones that contain multiple system calls, can be trivially implemented using this attacking technique.

2.2. Gadgets and Unintended Instructions

For ISAs such as x86 with variable size instructions, the attackers can find gadgets that are unintended by the programmer. Specifically, these are instructions that start at a byte in the middle of a multi-byte instruction. These instructions account for a large number of the gadgets exploitable by attackers [8].

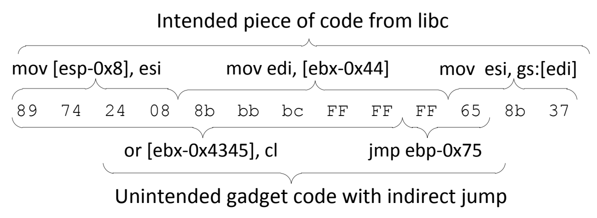
Intended piece of code from libc
mov [esp-0x8], esi
mov edi, [ebx-0x44]
mov esi, gs:[edi]
89 74 24 08 8b bb bc FF FF FF 65 8b 37
or [ebx-0x4345], cl
jmp ebp-0x75
Unintended gadget code with indirect jump

Figure 3. Example Gadget with Unintended Jump

To illustrate the concept of unintended branches, we show a sequence of bytes from the libc library in the top part of Figure 3. If the decoding starts after skipping the first four bytes, a different instruction sequence can be decoded as shown at the bottom of Figure 3, containing an indirect jump that the programmer did not intend to execute. We show later that these unintended gadgets make up the majority of the gadgets available for the attackers, significantly improving the attackers ability to mount attacks. However, although unintended gadgets far exceed intended gadgets in number, they are often harder to utilize because they can include rarely-used instructions with complicated addressing modes and constants. Thus, only short unintended gadgets are typically usable.

3. Understanding Signatures of JOP Attacks

Signature based defenses can only work if the instruction patterns exhibited by the attack code can be distinguished from those of normal programs. The JOP attack patterns (the number and length of gadgets used) are different from the patterns of ROP attacks examined in [13] because of two factors: 1) the reliance on indirect jumps instead of returns; and 2) the need to execute the dispatcher gadget to orchestrate the gadget-level control flow, thus requiring more gadgets for an attack.

In terms of the number of gadgets, Chen et al. [13] reported that at least three consecutive gadgets are needed to carry out even a simple ROP attack. For JOP, the number of gadgets needed is higher because of the need to call the dispatcher gadget after every functional gadget. In addition, it is much easier to compose an attack using short-length gadgets to limit the undesirable side effects on the program state.

All existing tools for automatic gadget discovery [8, 47] therefore limit the gadget size to at most five instructions and only consider usable the gadgets that perform one operation (and one state update). The work of [13] also used gadget sizes of at most five instructions for implementing the shellcodes in ROP-style attack. Signature based detection relies critically on these threshold values, so it is important to verify that they hold.

3.1. Gadget Analysis for JOP Attack

The size of a usable gadget is limited by the side-effects that the gadget has on the program state (including memory locations and registers). Large gadgets typically overwrite many registers and/or memory locations, thus corrupting the state and making attack continuation very difficult or impossible. This is especially true for the gadgets that are comprised of unintended instructions.

To understand the side-effect properties of the JOP gadgets, we performed extensive gadget analysis within the code base of several libraries. Our gadget discovery algorithm starts with building the gadget trie as described by Shacham et al. [47]. In a gadget trie, indirect jump instructions are represented as nodes immediately under a dummy root node. A child node under an indirect jump represents a possible decoding of an instruction preceding the parent instruction. Since multiple possible instructions (all but one unintended) can precede an indirect branch, the trie can branch leading to multiple gadgets ending at the same indirect branch. Once the trie is constructed, the algorithm traverses the nodes starting with an indirect branch toward its children, and every path along this traversal represents a possible gadget.

Signature detection relies critically on the observation that usable gadgets are short allowing us to distinguish attacks from normal programs where the distance between indirect branches are significantly longer. We base our approach to the usability of gadgets on the number of state updates that a gadget performs. State updates are register limiting instructions such as register writes or indirect memory accesses (which force registers to be a specific value in order to prevent illegal accesses). We contend that longer gadgets that make multiple state updates are difficult to use without destroying the attack state.

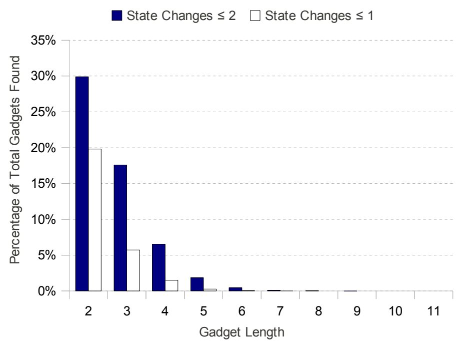

Figure 4. Libc statistics on gadget length and state changes

Figure 4 shows the total number of gadgets discovered by the algorithm in the standard C library (libc), as well as the number of gadgets that remain after we remove the gadgets that do more state changes than each given threshold. Figure 5 shows the same gadget statistics for other common libraries. The top part of the figure shows the total number of gadgets of a given length (each length is a separate figure). The bottom part shows the number of gadgets present (of the same length as the corresponding top figure) with at most one state update. While a significant number of gadgets of various sizes obviously exist in the libraries, there are no gadgets of size eight instructions or more that perform less than two state updates (to memory or registers).

Figure 6 shows the average number of side effects as the gadget length increases. It also shows the minimum number of side effects in gadgets of that length found across all the libraries we studied. As the gadget length grows the number of side effects grows linearly making them increasingly more difficult to use.

Even at a threshold of 7, there exists only one gadget with a single state update in libc, and another one in libglib-2.0. Upon further examination, we found both of these gadgets not to be usable because they use unintended instructions that cannot be used. Since no suitable gadgets of seven instructions or more were found in multiple libraries, a threshold of seven instructions can be used by SCRAP to identify a gadget. However, using this length as a hard threshold represents a strong assumption: the attacker may be able to tolerate some of the side-effects in a long gadget, allowing her to use it as a delay gadget and bypass the detection. We later relax this assumption to build signature detectors that are resilient to the presence of some longer gadgets.

4. Stealth JOP Attacks: Concealing Attack Patterns With Delay Gadgets

From the discussion in the previous section, it appears that simple signature-based detection can be effectively applied to protect against JOP attacks. However, when designing security solutions it is important to assume that the attacker is aware of the particular defense that is implemented and consider possible attack modifications that would bypass this protection.

All JOP and ROP variations developed to date only considered the functional requirements of the attack. Therefore, all gadgets used by the attackers were performing some use-

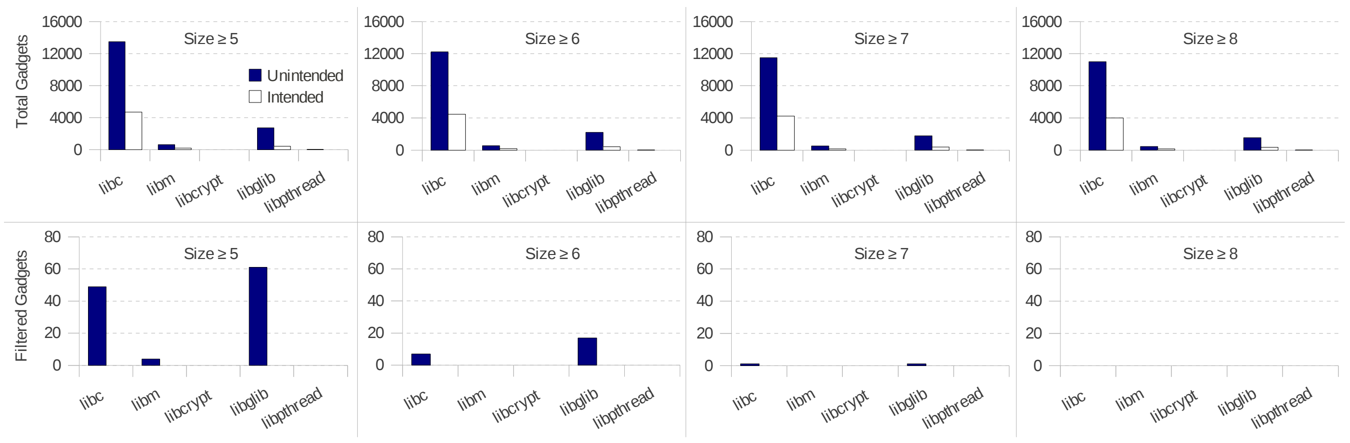

Figure 5. Gadget Length and Side Effect Analysis: Top figures show the total number of gadgets of a given length while the bottom figure shows the gadgets for the same length with the shown number of side effects.

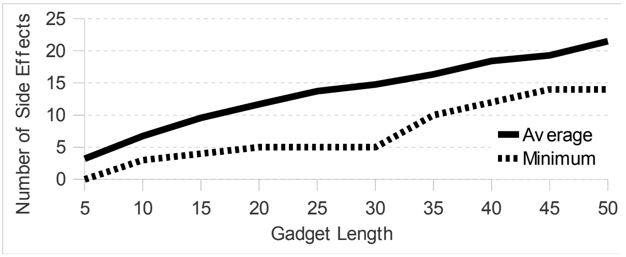

Figure 6. Number of Side effects as gadget length increases

ful part of the attack code. In addition, to avoid the necessity of dealing with gadget side-effects, the existing automatic tools for generating JOP and ROP attacks only consider small gadget sizes. Signature-based approaches are effective under these assumptions, as shown in [13] and also by the analysis in the previous section.

However, what if the attacker is aware of the signature-based protection and modifies the attack to distort its execution patterns from those expected by the defense? One approach for accomplishing this is to introduce a delay gadget in the middle of the attack. The purpose of a delay gadget is not to execute any part of the attack code, but rather perform some spurious computations in a way that would not corrupt the machine state needed by the attack. At the same time, the gadget would be long enough to reset the gadget count used by the signature detector, before an attack is detected. In this section, we introduce such delay gadgets and demonstrate how the attack shown in the background section can be modified to incorporate it.

The analysis in the previous section showed that long gadgets have too many side effects to be usable; however, it is possible to create a small sized delay gadgets by using a call to a function. Since most functions have no side effects, they represent an ideal vehicle for implementing delay gadgets without destroying the program state. If a function call results in executing a larger number of instructions the signature based attack detector will reset (assuming that this is a valid program), allowing the attacker to continue the attack. In the remainder of this section, we demonstrate how to implement a delay gadget using a function call (using `atoi()`).

	Gadget	Gadget Function

g3	`call, [ecx-0x56000a00]`
	`add bl, bh`
	`inc ebx`
	`add dh, bh`
	`jmp edi`
		Delay

Table 2. Delay Gadget Used in Stealth JOP Attack

An example of a delay gadget that makes a call to the *atoi()* function is shown in Table 2, this gadget was found in the libc library. `atoi()` executes many more instructions than the typical JOP gadgets, bypassing signature based detection. When `atoi()` returns, some registers such as `eax`, `ecx`, and `edx` may have been altered and do not contain data that is meaningful to the attack. However, by convention, other registers such as `ebx`, `esi`, `edi`, `esp`, and `ebp` are saved. As long as the delay gadget ends with an indirect jump based on one of these saved registers, the attack can return to the dispatcher gadget which can recover from any side effects caused by the delay.

This new attack, which we call Stealth-JOP, is mounted using the same series of gadgets as our previous example, but with delay gadgets called periodically to avoid detection. Our previous JOP attack jumped from the dispatcher gadget to a functional gadget, and then back to the dispatcher. The Stealth-JOP attack example jumps from the dispatcher to a functional gadget, and then to the delay gadget. After the delay gadget has executed, the control returns to the dispatcher. Thus, there is no sequence in the code with multiple consecutive short gadgets, making DROP-like signature detection fail. At the same time, the attacker is able to execute arbitrary code using the short functional gadgets.

In addition to considering delay gadgets through function calls, it is important to note that if even one gadget of length higher than the detection threshold in DROP can be used (or at least tolerated) in an attack, then an attacker can exploit this gadget to bypass signature detection. We build the basic SCRAP detectors first assuming that the gadget lengths derived in Section 3 represent hard limits; that is, every gadget that makes 2 side effects or more is not usable. However, it is highly likely that a motivated attacker will be able to find at least some longer gadgets whose side effects can be tolerated; we were able to identify multiple such gadgets in constructing our attacks. We first develop signature detectors assuming these hard limits on gadget lengths. However, we then relax this assumption and develop more sophisticated signature detectors that are able to tolerate the presence of some longer gadgets and still detect an attack.

5. Threat Model, Assumptions and Limitations

We use standard CRA assumptions on the attacker's access to memory; this could be obtained using a buffer overflow, a string formatting attack, or a non-local jump buffer (using `setjmp` and `longjmp` [30]). We assume that the system has NX support for writable memory such that code injection attacks are not possible.

We assume that the attacker can find arbitrary gadgets limited only by the attack lengths as per the analysis we showed in Section 3. Later we relax this assumption by allowing the use of longer gadgets. Throughout the paper, we present real attacks constructed from existing library code. However, rather than assume security due to our inability to find gadgets in the current version of the libraries, we make the assumption of the existence of arbitrary gadgets such that the defense works with any future code base, and not just the ones we used for the analysis.

We assume that the vulnerability exploited to initiate the attack does not lead to a privilege escalation. If privilege escalation is achieved from the initial vulnerability, then a CRA attack is not necessary. The attacker may seek to obtain privilege escalation through the CRA.

The new stealth JOP attack proposed in this paper uses delay gadgets to obfuscate the JOP execution pattern. We explored the use of function calls as delay gadgets because of the limited side-effects that they generate. Our analysis also showed traditional gadgets are ineffective beyond a certain length because of the presence of state updates. However, there is a possibility that additional patterns of generating delay gadgets may exist (e.g., a loop gadget), although we have not been able to find and exploit such gadgets. We believe that the detection logic can be extended to capture such delay patterns as well.

6. Expressing Attack Signatures in Formal Language

In this section, we formalize the attack pattern as a context-free grammar. This formal description is used as the basis for the hardware implementation of SCRAP logic. We encode executions of instructions as strings of symbols denoting types of instructions, called signatures. The attacks are then formalized as formal languages of signatures. The alphabet used in this section is given in Table 3.

6.1. Expressing Attacks Without Delay Gadgets

We observe that basic CRAs, such as ROP and JOP attacks, can be expressed as a formal language defining an attack as the following regular expression that uses POSIX Extended Regular Expressions:

Symbol	Instruction
w	Indirect Jump
x	Indirect Call
y	Call
z	Return
a	All Other

Table 3. Signature Alphabet

$$
\begin{aligned}
V =\{&Attack, P, Gadget, Delays, Delay,\\
&Call, Body, Return, Gadget, Indirect,\\
&NotGadget, NotAttack\}\\
\Sigma =\{&w,x,y,z,a\}\\
Rules =\{&\\
Attack \to{}& P\ P\ P\\
P \to{}& Gadget\ Delays \mid Delays\ Gadget\\
Gadget \to{}& Indirect \mid a\ Indirect \mid a\ a\ Indirect \mid\\
& a\ a\ a\ Indirect \mid a\ a\ a\ a\ Indirect \mid\\
& a\ a\ a\ a\ a\ Indirect\\
Indirect \to{}& w \mid x\\
Delays \to{}& Delay\ Delays \mid \varepsilon\\
Delay \to{}& Call\ Body\ Return\\
Call \to{}& x \mid y\\
Return \to{}& z\\
Body \to{}& Delays\ Body \mid Body\ Delays\\
Body \to{}& a\ Body \mid Body\ a \mid \varepsilon\\
Body \to{}& NotGadget\ NotAttack\\
NotGadget \to{}& a\ a\ a\ a\ a\ a\ Indirect \mid a\ NotGadget\\
NotAttack \to{}& \varepsilon \mid P \mid P\ P\}
\end{aligned}
$$

Figure 7. Definition of $G_{5,3} = (V, \Sigma, Rules, Attack)$

$$R_{N,S} = (a\{0,N\}\,(w|x))\,\{S,\}$$

Here, w denotes an indirect jump and x denotes an indirect call, while a denotes any other type of instruction. N is a parameter that specifies the number of instructions that a gadget can have, while S specifies the number of consecutive gadgets considered as an attack. For example, in $R_{5,3}$ case, three consecutive gadgets each having no more than five instructions form an attack.

6.2. Expressing Attacks with Delay Gadgets

With the inclusion of function calls as delays, the formal language defining the attack becomes a context free language, formalized as the context-free grammar $G_{N,S}$, where again N is the number of instructions that a gadget can have and S is the number of consecutive gadgets considered as an attack. The definition of $G_{5,3} = (V, \Sigma, Rules, Attack)$ is given in Figure 7.

The grammar starts with *Attack* which is expanded to $S = 3$ phases, each including a gadget and an unbounded number of delays. A gadget is the same as the $G_{N,S}$ regular expression defined above in Section 6.1. A delay starts with a *Call* and ends with a *Return* and a *Body* between them which we further define to capture complex delay gadgets consisting of nested function calls. Specifically, the delay gadget can have any number of delay function calls, and any number of unimportant instructions. It can also include less than S gadgets in it as long as there is a *NotGadget* sequence before it. A *NotGadget* has more than N instructions before the *Indirect* instruction.

The grammar is given for specific N and S values, but it can be reformulated for any N and S value by simply changing some of the production rules. *Attack* has S number of P expansions and *Gadget* allows N many a's before *Indirect*. *NotGadget* and *NotAttack* would also have to be changed accordingly.

Signature	$\in \mathbf{R_{5,3}}$**?**	$\in \mathbf{G_{5,3}}$**?**
aaawaawaaw	Yes	Yes
awaaxaaaaw	Yes	Yes
awaxaaaaaazaxaw	No	Yes
awaxaayaazazaxaw	No	Yes

Table 4. Example Attack Signatures

Table 4 shows example attack signatures and whether they are considered as an attack under prior approaches described in Section 6.1 and under the grammar that excludes delays. The parts of the signature that are matched as delays under $G_{5,3}$ are highlighted.

7. SCRAP: Hardware-based Signature Detection

In this section, we demonstrate an efficient hardware implementation to recognize the formal grammar that expresses the attack signatures shown in the previous section. The proposed logic required by SCRAP involves less than 200 bytes of storage (for a 4-way superscalar processor) and is located at the commit stage of the pipeline off of the critical timing path. In the subsections below, we describe the components of SCRAP, building from a single state machine towards developing the complete solution. This is a standard exercise of translating the language grammar into the hardware implementation; however, because up to four instructions commit every cycle, we introduce an optimization that significantly simplifies the logic without having any adverse impact on the performance.

7.1. SCRAP State Machine

The SCRAP state machine is shown in Figure 8. We use a counter to keep track of the current gadget length, and a comparator to decide whether it is above the gadget length threshold. When a gadget end is detected (w or x event in the language), the gadget length is used to transition through the shown finite state machine. The remaining step to implement the push down automata is to note that when a call instruction is encountered, we push the current state number to the shadow stack. This number is restored when a return instruction is encountered.

7.2. Integrating State Counters into Secure Call Stack

As we discussed previously, a shadow call stack is a mechanism that has been proposed to defend against simple ROP attacks [32, 35, 49, 58]. Both software and hardware implementations of this stack have been developed. SCRAP

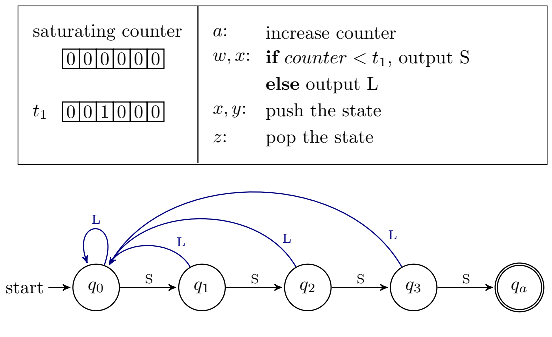

Figure 8. State Machine for SCRAP

relies on a hardware implementation of the call stack, which is backed up by a a software stack. In our design, each entry of the hardware stack is augmented with the counter that keeps track of the number of potential attack gadgets that executed consecutively. This makes it possible to track the information about the state of the attack even across function calls, eliminating their use as delay gadgets.

7.3. The SCRAP Microarchitecture

We now describe the microarchitectural changes needed for an out-of-order superscalar processor to implement SCRAP. First, as the instructions are decoded, the information about the relevant instruction types is extracted and placed in the Reorder Buffer (ROB) entries allocated for the instructions. For this purpose, all instructions are classified into five types, as defined by the attack grammar in Section 6, thus requiring a new 3-bit wide field within each ROB entry to carry this information. When the instructions reach the commit stage of the pipeline, this information is used to update the SCRAP state machine counters.

The complexity of the counter update logic depends on the superscalar width (i.e. how many instructions commit per cycle) and also on the thresholds on the gadget length and the number of consecutive gadgets used by SCRAP. To simplify the logic, to ensure that only one counter update can be performed per cycle, and also to ensure that in a single cycle we operate on the counters within a single entry of the secure stack, we propose a technique called Commit Throttling.

7.3.1. Simplifying SCRAP through Commit Throttling

To simplify the SCRAP state machine counter update logic, we propose Commit Throttling (CT), which allows only one of the following instructions to be committed in a single cycle: CALL, indirect CALL, indirect jump, and RET. The number of these instructions in typical programs is small (less than 5% combined according to our analysis based on the binary instrumentation of SPEC 2006 benchmarks). When encountering the second instruction from this list in the co-committing group in the same cycle, the commit logic blocks and delaying the commit the second instruction to the next cycle. An additional requirement that we impose is that whenever a return instruction is encountered, the commit process also stops to ensure that we always operate on the counters within the same stack entry in each cycle. The impact of CT optimizations on the performance is negligible (less than 0.03% on the average for SPEC 2006 benchmarks), but it allows us to significantly limit the number of different instruction patterns coming out of the commit stage in a single cycle in terms of their impact on the SCRAP detection state.

We implemented the SCRAP logic in Verilog HDL on a Xilinx Spartan-3E XC3S100E FPGA with a 90nm process, using Xilinx ISE WebPACK 14.1. The delay of the SCRAP logic is 3.51 nanoseconds, allowing a clock frequency of 284 MHz. For comparison, the delay is 2.97 ns for an 8-bit counter update logic and 3.35 ns for a 16-bit one. The FPGA implementation shows that the delay is well under the cycle period of a superscalar processor.

7.4. Allowing software configuration of SCRAP

We allow the SCRAP detector thresholds to be configurable using a privileged system call that sets the detection machine state. We build large detector allowing up to 10 gadgets in a row to be detected. The configuration can be changed to $G_{x,y}$ by changing the t_1 threshold register to x and by marking the y^{th} state in the detector to be the finish state detecting the presence of an attack.

The choice of software configurability is made for two reasons. First we observed significant divergence in application behavior. Without software configurability, we are forced to use the worst case thresholds that do not generate false positives across any applications. Many applications do not use indirect branch and call instructions frequently, and can benefit from lower thresholds which further increase the difficulty of attacks. At the same time, we want to protect against the potential of an application that does generate false positives against our thresholds. If the thresholds are fixed in hardware, then such an application cannot be supported. Allowing the thresholds to be changed, or the detector to be disabled (for example, for applications that do not communicate with untrusted parties), supports these cases in a manner similar to how the NX bit can be disabled by software.

8. Performance Evaluation of SCRAP

For evaluating the performance impact of SCRAP, we used PTLsim [59] - a cycle-accurate x86 processor simulator. We simulated a 4-wide issue out-of-order core with 64KB L1 data and instruction caches, 512KB L2 cache and 2 MB L3 cache. Memory latency was assumed to be 100 cycles. We used 18 C and C++ SPEC CPU2006 [51] benchmarks for our experiments. The benchmarks were compiled using GCC-4.2 compiler on a x86 machine running Ubuntu with kernel version 2.6.24.

Each benchmark was simulated for 2 billion committed instructions after fast-forwarding for the first 100 million instructions.

First, we studied the impact of the Commit Throttling optimization. We discovered that there was negligible slowdown due to CT (less than 0.1% on average). To explain this slowdown, we show in Figure 9 the percentage of cycles where CT initiated a commit block. The cost of most of these stalls is hidden by out-of-order execution, resulting in the observed low impact on overall performance.

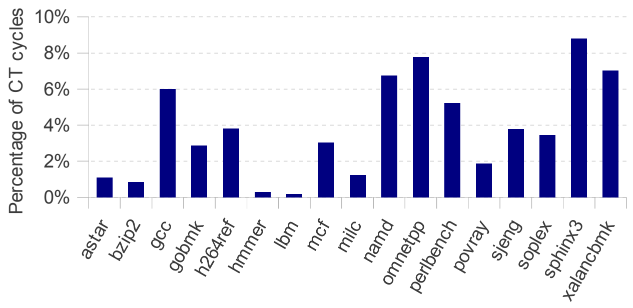

Figure 9. Percentage of cycles where commit is blocked by CT

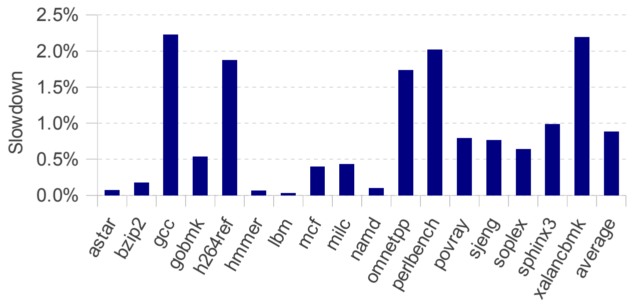

Figure 10. Performance Slowdown of SCRAP

For a 4-entry hardware call stack, the performance overhead of SCRAP is just over 1% on the average and it is less than 6% for all benchmarks as shown in Figure 10. This includes the overhead of stalls due to CT cycles as well as the overhead of the overflow of the hardware shadow stack. We used a hardware shadow stack of size four entries.

9. Security Analysis of SCRAP

In this section, we analyze the SCRAP detection effectiveness. We first demonstrate that it results in no false positives for normal programs and then analyze detection of actual shellcodes.

9.1. False Positives in Regular Codes

Next, we examine the impact of SCRAP on the execution of real programs to determine if SCRAP generates any false alarms during legal program execution. The results are presented for SPEC 2006 benchmarks in Figure 11 and Figure 12. The benchmarks (the entire suite) were instrumented using Pin tool and were run to completion. As seen from these figures, the rate of false alarms depends on the selected thresholds. We observe that for the thresholds with three consecutive gadgets and at most seven instructions in each gadget, none of our benchmarks generated false positives; i.e., a SCRAP detector $G_{7,3}$ generates no false positives.

We also performed experimental evaluation on Apache 2 Web Server for 28 threads, executing one billion instructions in total, while Apache is trying to serve a static webpage of about 65KBs to the ab tool which is sending thousands of requests running remotely. We performed a similar evaluation for Mozilla 14.0.1 with 18 threads/processes, also for a total of one billion instructions, while it was trying to access a Wikipedia entry. Apache had no false positives for $G_{<9,>=3}$. Even though Firefox started showing false positives for $G_{7,3}$, there were no false positives for $G_{7,>3}$ or $G_{<7,>=3}$.

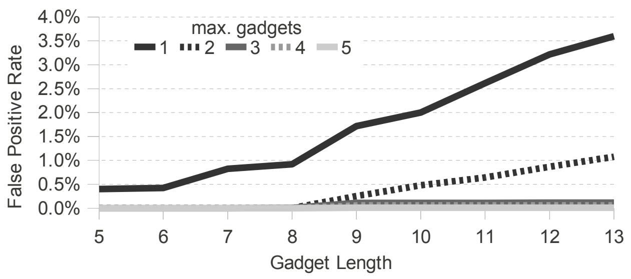

Figure 11. False positive rates for different number of consecutive gadgets

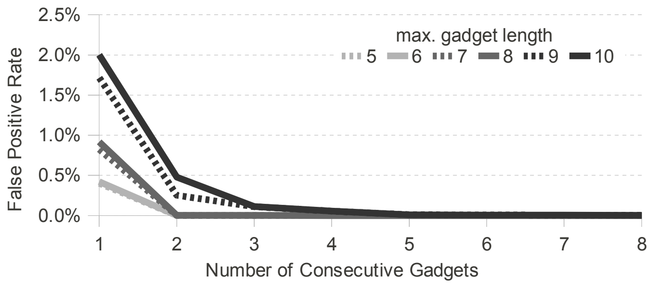

Figure 12. False positive rates for different gadget lengths

9.2. Detecting JOP Attacks

With a SCRAP detector $G_{7,3}$, SCRAP is capable of detecting any JOP attack that does not use a gadget of length 7 or more. Thusfar, every published attack, and every attack automation tool uses gadgets of size 5 or less [13, 25, 47]. As seen in Section 4, gadgets that call functions can be used in an attack because they preserve half of the registers due to assembly convention. However, SCRAP is capable of detecting attacks that implement these gadgets, while a JOP version of DROP would fail. As discussed in Section 3, in general long gadgets that do not use function calls have too many side effects to be used in an attack. Therefore, all currently published attacks would be detected by SCRAP. However, if an attacker is aware of the SCRAP protection, they may be able to find longer gadgets whose side effects can be tolerated or repaired by a subsequent gadget. Thus, we extend SCRAP in Section 10 to defend against such possible JOP attacks that manage to use an occasional long gadget in the middle of the attack to avoid detection.

To further assess SCRAP detection capabilities, we implemented 140 shell code attacks available from the Shell-Storm Linux shellcode repository [48]. These shellcodes ranged in complexity from simple single system calls, to attacks with multiple system calls, conditional branches, and loops. Even the most basic attack required at least 6 gadgets, which is greater than the minimum number of consecutive gadgets necessary to be detected by SCRAP. Gadgets longer than 6 instructions were extremely difficult to incorporate due to side effects. However, we were able to include a small number of gadgets of intermediate length, a few instructions longer. Attacks that use these longer gadgets are defeated by the improved detector presented in Section 10.

10. Tolerating Longer Gadgets

Thusfar, we have assumed that the length of the gadgets usable by attackers is limited to a hard threshold chosen in a way that makes false positives impossible. This assumption is based on the analysis in Section 3 where we showed that longer gadgets create too many state updates, making them difficult to use (e.g., Figure 6). However, it may be possible for attackers to identify some longer gadgets whose side effects do not completely destroy the attack state. Such gadgets can be used as a delay gadget to avoid detection by the basic SCRAP detector. In our own implementation of shellcodes, although it was difficult, we were able to identify a few such gadgets that are longer than the detection threshold and could be integrated into an attack successfully, avoiding detection by the basic SCRAP. These gadgets, for example, updated registers that were not needed for the attack, modified a non-critical memory location while being able to avoid illegal accesses, or had a side-effect that could be undone by another gadget. Thus, for practical signature based detection, it is imperative that we detect attacks even in the presence of some of these longer gadgets.

In the remainder of this section we propose a new multi-threshold detector that is able to detect CRAs quickly, while tolerating the use of longer gadgets. Intuitively, the detector assumes that attackers may be able to find some gadgets longer than the SCRAP threshold whose side-effects can be tolerated or undone by subsequent gadgets. These intermediate gadgets are not easy to find or use constructively in an attack since the number of side effects made by a gadget grows quickly with the length of the gadget. Side effects also increase the number of gadgets necessary for an attack; a repair gadget must be called in order to correct state changes and a dispatcher gadget must be called in order to reach the repair gadget.

The new detector detects attacks as a sequence of gadgets of length t_1 or shorter, while allowing the use of intermediate gadgets (IGs) of length t_2 or shorter such that $t_2 > t_1$. Since IGs typically do not advance the attack but are used only to avoid detection, we do not advance the gadget count (move closer to detection) like we do with short gadgets. At the same time we only reset to the initial state with gadgets of length greater than t_2. Now, for every other IG the gadget counter is reduced by one to take advantage of the additional gadgets necessary to repair side effects. To detect an attack, we still need k short gadgets($<t_1$) before a very long gadget ($>t_2$). The state machine for the multi-threshold detector is shown in Figure 13. We call a detector of this type $G_{t_1,t_2,l}$ where l is the gadget count that is needed to detect an attack. Note that all three thresholds are software configurable in privilege mode.

The false positive rate is increased by this new multi-threshold detector. Previously, medium length gadgets reset

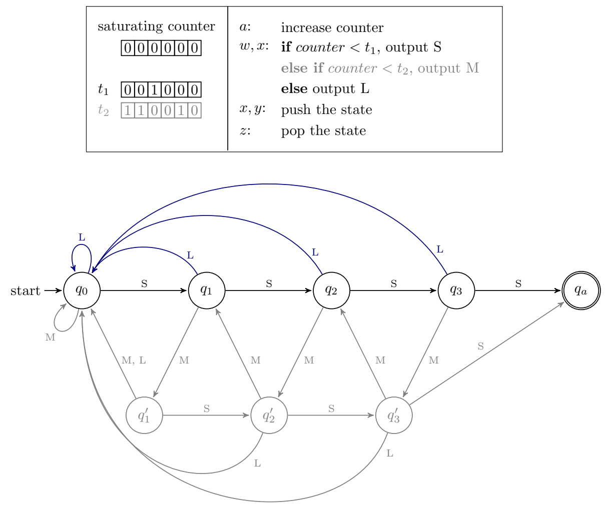

Figure 13. State Machine for the two threshold Detector

SCRAP to its initial state, setting all counters to 0, making it more difficult to detect an attack (but making it possible for attackers to avoid detection). Table 5 shows the total false positive rate for the entire SPEC 2006 benchmark suite. The results show that t_1 can be set to 8, and t_2 can be set to a very high length of 50 without any false positives with gadget count, l, of 7. Gadgets of length 50 in the libraries we examined have a minimum of 15 side effects and an average of 20 side effects (Figure 6)–it is extremely improbable that they can be used without destroying the critical attack state.

l	8-10	8-15	8-20	8-25	8-50
1	1136837	3183299	2148070	3147423	12403895
2	335186	1446333	499522	517931	631640
3	4022	494252	129540	165565	186011
4	0	116485	4	2478	2837
5	0	1	1	1	31
6	0	0	0	0	8
7	0	0	0	0	0

Table 5. False positives for with varying detection thresholds t_1 and t_2

As a further enhancement, a simple $G_{7,3}$ SCRAP module, as discussed in Section 9, could be used concurrently with this multi-threshold detector to catch attacks that use three short gadgets in a row. The overhead of this approach is linear in the number of detectors since a new state machine has to be implemented for each detector, and a space on the stack is needed to save each detector's state upon a function call.

We also evaluated latency for a two threshold detector using ten bits for counters (to allow reconfiguration of l up to a value of 10). The logic can be clocked at up to 222 MHz with a period of 4.5 nanoseconds in the FPGA process we used as a target. For comparison, a 32-bit counter update logic in the same process has a delay of 4.52 ns.

11. Related Work

In this section, we overview different approaches to protecting against CRA attacks. The related work is organized into three parts: (1) defenses against buffer overflow attacks; (2) comprehensive defenses; and (3) defenses specific to Code Reuse Attacks (CRAs).

11.1. Defenses against Buffer Overflows

Buffer overflows are one of the most common software exploits in languages without type safety such as C/C++. A buffer overflow is necessary to initiate the CRA attack. Several approaches were developed to defeat buffer overflows [6, 15–17, 26, 56]. Stackguard [17] and ProPolice [26] are GCC extensions that use canaries. StackShield separates return addresses into a separate stack at compile time making it impossible for stack buffer overflows to overwrite the return address [56]; similar works save a copy of the return address and validate it before a function return [6, 15]. All the preceding approaches require compiler support and cannot protect legacy binaries. They also cannot prevent heap overflow attacks or attacks on function pointers [43].

Hardware solutions have been proposed to protect against stack smashing. StackGhost uses the register window feature of the Sun Sparc architecture to verify that return addresses have not been overwritten [28]. Recently, the advent of the NoExecute (NX) bit and its support by mainstream operating systems have made code injection attacks ineffective [4, 54].

11.2. Comprehensive Defenses

Memory bounds checking (MBC) annotate pointers with their legal address range and check every memory access against the base and bound of the associated data structure [22, 23, 29, 36]. However, the overhead of MBC is substantial. MBC cannot prevent all memory exploits: it cannot protect legacy binaries and externally linked or loaded components. It is difficult to track legal memory bounds around memory aliasing, variable length function argument lists, complex data types and other programming language constructs.

Dynamic Information Flow Tracking (DIFT) taints the information coming from insecure sources, and dynamically tracks and propagates the taint through processor registers and memory locations. If a tainted address is used for writing into the stack, a security exception is raised. The drawback is that DIFT is a heavy-weight approach that entails a significant redesign of the processor datapath and memory system if implemented in hardware [18, 42, 52], or incurs a substantial performance overhead if implemented in software [39, 45]. In addition, DIFT solutions may suffer from false positives, where the tainted state of the system rapidly expands in a domino fashion.

Data flow integrity [10] derives the data flow graph during compile-time and instrument the program to enforce conformance with the flow in the graph; note that this is a dual approach to control flow integrity. Using similar analysis, WIT [2] associates instructions with their allowed target objects and enforces integrity of each write operation.

11.3. CRA Attacks and Defenses

The first CRA attack proposed was the *return-into-libc (RILC) attack* [21], where the attacker subverts the control flow to call a function in the standard C library. Extensions to basic RILC have been proposed to allow a static chain of libc functions to be called [37] and recently to allow a general data-dependent form of chaining of libc functions [55]. With the exception of the last attack [55] which relies on a form of jump oriented programming described below, RILC attacks cannot support arbitrary computation on the victim machine.

Return-oriented Programming (ROP) attacks were recently proposed [47], and the number of solutions to them were introduced [13, 33, 35, 49, 58]. We discussed those solutions in detail in earlier sections of this paper.

The newer defenses against ROP attacks also attempt to address JOPs. For example, Onarlioglu et al. first use binary rewriting to remove unintended branches and returns [41]. To protect intended branches, they use function-specific markers on each stack frame; they call these markers stack cookies. They also insert checks after every branch to check the stack cookie. However, if gadgets are available in a function to replace the cookie before leaving, this protection may be defeated. Because it requires binary rewriting, the approach cannot easily protect legacy binaries; it also increases the code footprint by over 25%.

Kayaalp et al [32] propose branch regulation, a hardware supported techniques to protect against JOPs. Using binary rewriting, they insert markers at the beginning of every function, which include a magic number to mark a legal function entry, as well as the length of the function. Branch regulation requires binary rewriting and cannot easily protect legacy binaries. It is also possible that a function may exist that can provide sufficient gadgets to mount an attack; security is not completely guaranteed.

Control flow integrity [1] is an approach to enforce legal control flow inside of programs; CFI would identify the illegal control flow necessary for code reuse attacks. CFI requires deep analysis of the source or binary to derive the CFG and substantial overhead to track the control flow of the program.

Address space layout randomization (ASLR) [53] randomly offsets the program location in memory. ASLR and other optimized heap allocation models [7, 57] hide the correct address of the malicious code hiding the location of the gadgets. Unfortunately, exploits against ASLR are known; for example, a a format string attack can expose the stack location to an attacker allowing the random offset to be derived [38]. Schwartz et al show that even a small part of the code being unrandomized is sufficient to construct CRA attacks [25].

12. Concluding Remarks

In this paper, we presented SCRAP, a new hardware-based architecture for protecting against the emerging class of code reuse attacks (CRAs). We demonstrated that the latest incarnation of CRAs - jump oriented programming (JOP) attacks - have execution patterns that are clearly distinguishable from the patterns exhibited by regular programs. However, we also demonstrated a new attack that renders previously proposed signature-based approaches ineffective by introducing delay gadgets. Delay gadgets are gadgets whose only purpose is to obfuscate the execution patterns of the attack without performing any useful computation. We developed a complete working JOP attack that incorporates delay gadgets. We then proposed and developed the SCRAP architecture for efficiently detecting such stealth JOP attacks. Our design started from the development of the formal language describing the stealth JOP attack signature and then subsequent demonstration of the hardware implementation. We also proposed a new microarchitecture optimization to simplify the SCRAP logic without encountering any performance loss.

In summary, SCRAP architecture protects unmodified legacy binaries, involves no changes to the software layers and incurs very small performance degradation: less than 2% on the average across the SPEC 2006 benchmarks. We also show that with appropriate selection of the detection thresholds, SCRAP successfully detects all JOP attacks used to implement many existing shellcodes, but at the same time results in no false alarms for the regular applications. It is therefore an effective and low-overhead protection, which in the very least increases the attack complexity dramatically, if not making it completely impossible. SCRAP can be configured in software to allow the thresholds to adapt to applications or to disable protection if it is not desired.

We extended the basic SCRAP detector to allow it to tolerate the presence of longer gadgets using a two threshold detector. The new detector can tolerate the presence of intermediate gadgets of length up to 50 instructions, without generating any false positives on the SPEC 2006 benchmark. We implemented both SCRAP and the two-threshold SCRAP in a hardware description language; the required hardware is small and fast. It also resides at the commit stage and does not affect any of the critical pipeline stages.

13. Acknowledgements

We would like to thank the anonymous reviewers for their comments and suggestions. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-09-1-0137 and by National Science Foundation grants CNS-1018496 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of CCS*, pages 340–353. ACM, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. *Security and Privacy, IEEE Symposium on*, 0:263–277, 2008.

[3] Aleph One. Smashing the stack for fun and profit, Nov. 1996.

[4] S. Andersen. Part 3: Memory protection technologies. In *Changes to Functionality in Microsoft Windows XP Service Pack 2*. Microsoft Corp., 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[5] W. Baer and A. Parkinson. Cyberinsurance in IT Security Management. *IEEE Security and Privacy*, 5(3):50–56, may/june 2007.

[6] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conf.*, pages 251–262, 2000.

[7] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of PLDI*, pages 158–168. ACM, 2006.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of ASIACCS*, pages 30–40. ACM, 2011.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of CCS*, pages 27–38. ACM, 2008.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of OSDI*, pages 147–160. USENIX Association, 2006.

[11] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of CCS*, pages 559–72. ACM Press, oct 2010.

[12] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the avc advantage. In *Proceedings of EVT/WOTE*. USENIX/ACCURATE/IAVoSS, aug 2009.

[13] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of ICISS*, pages 163–177. Springer-Verlag, 2009.

[14] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of ASIACCS*, pages 20–29. ACM, 2011.

[15] T. cker Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS'01*, 2001.

[16] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of USENIX Security*, pages 7–7. USENIX Association, 2003.

[17] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of USENIX Security*, volume 7, 1998.

[18] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of ISCA*, pages 482–493. ACM, 2007.

[19] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-Oriented Programming without returns on ARM. Technical report, System Security Lab - Ruhr University Bochum, 2010.

[20] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of ACM STC*, pages 49–54. ACM, 2009.

[21] S. Designer. "return-to-libc" attack, 1997. Bugtraq.

[22] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *Proceedings of the ASPLOS*, pages 103–114, New York, NY, USA, 2008. ACM.

[23] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of ICSE*, pages 162–171. ACM, 2006.

[24] T. Dullien and T. Kornau. A framework for automated architecture-independent gadget search, 2010.

[25] T. A. Edward J. Schwartz and D. Brumle. Q: Exploit hardening made easy. In *Proceedings of USENIX Security*, 2011.

[26] H. Etoh and K. Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIG notes on computer security*, Oct 2001.

[27] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of CCS*, 2008.

[28] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of USENIX Security*, pages 5–5. USENIX Association, 2001.

[29] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman. Architectural support for low overhead detection of memory violations. In *Proceedings of DATE*, 2009.

[30] T. O. Group. IEEE Std 1003.1, 2004. http://pubs.opengroup.org/onlinepubs/009695399/functions/setjmp.html.

[31] R. Hund, T. Holz, and F. C. Freiling. Returnoriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proceedings of Usenix Security*, 2009.

[32] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low overhead mitigation of code reuse attacks. In *Proceedings of ISCA*, 2012.

[33] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of EuroSys*, pages 195–208, New York, NY, USA, 2010. ACM.

[34] F. Lindner. Developments in cisco ios forensics. confidence 2.0. presentation, 2009. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[35] J. McGregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. In *Proceedings of ITRE*, pages 243 – 250, aug. 2003.

[36] S. Nagarakatte, M. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of ISCA*, 2012.

[37] Negral. The advanced return-into-lib(c) attacks, 2001. http://www.phrack.org/issues.html?issue=58&id=4. Retrieved June 2012.

[38] T. Newsham. Format string attacks, September 2000. http://julianor.tripod.com/bc/tn-usfs.pdf.

[39] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS*, feb 2005.

[40] NIST national vulnerability database, 2012. Available online at http://nvd.nist.gov.

[41] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. Gfree: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of ACSAC*, pages 49–58, 2010.

[42] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of CF*, May 2011.

[43] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2:20–27, July 2004.

[44] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conf.*, pages 211–224, 2003.

[45] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of MICRO*, pages 135–148. IEEE Computer Society, 2006.

[46] R. G. Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, University of California, San Diego, 2009.

[47] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS*, pages 552–61. ACM Press, Oct. 2007.

[48] The shell storm linux shellcode repository, 2012. Accessed Sept. 2012 at http://www.shell-storm.org/shellcode/shellcode-linux.php.

[49] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[50] SOGETI ESEC R&D Lab. Analysis of the jailbreakme v3 font exploit, 2012. Retrieved September 2012 from http://esec-lab.sogeti.com/post/Analysis-of-the-jailbreakme-v3-font-exploit.

[51] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.

[52] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, pages 85–96. ACM, 2004.

[53] P. Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[54] P. Team. Pax non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt.

[55] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of RAID*, Sept. 2011.

[56] Vendicator. Stack shield technical info file v0.7, January 2001. http://www.angelfire.com/sk/stackshield/.

[57] O. Whitehouse. An analysis of address space layout randomization on windows vista, 2007.

[58] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *Proceedings of Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[59] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of ISPASS*, pages 23–34, 2007.