Defeating ROP Through Denial of Stack Pivot

Aravind Prakash*
Binghamton University
aravind@cs.binghamton.edu

Heng Yin
Syracuse University
heyin@syr.edu

ABSTRACT

Return-Oriented Programming (ROP) is a popular and prevalent infiltration technique. While current solutions based on code randomization, artificial diversification and Control-Flow Integrity (CFI) have rendered ROP attacks harder to accomplish, they have been unsuccessful in completely eliminating them. Particularly, CFI-based approaches lack incremental deployability and impose high performance overhead – two key requirements for practical application. In this paper, we present a novel compiler-level defense against ROP attacks. We observe that *stack pivoting* – a key step in executing ROP attacks – often moves the stack pointer from the stack region to a non-stack (often heap) region, thereby violating the integrity of the stack pointer. Unlike CFI-based defenses, our defense does not rely on the control-flow of the program. Instead, we assert the sanity of stack pointer at predetermined execution points in order to detect stack pivoting and thereby defeat ROP. The key advantage of our approach is that it allows for *incremental deployability*, an Achilles heel for CFI. That is, we can selectively protect some modules that can coexist with other unprotected modules. Other advantages include: (1) We do not depend on ASLR – which is particularly vulnerable to *information disclosure* attacks, and (2) We do not make any assumptions regarding the so called "gadget". We implemented our defense in a proof-of-concept LLVM-based system called `PBlocker`. We evaluated `PBlocker` on SPEC 2006 benchmark and show an average runtime overhead of 1.04%.

1. INTRODUCTION

With the advent of hardware mechanisms that prevent data execution (e.g., DEP, NX), attacks that reuse existing code are on a rise. Particularly, Return-Oriented Programming (ROP) [32] gleans code fragments terminated by `ret` instruction (or more broadly, an indirect branch instruction) called "gadgets" from executable sections of program code, and chains such gadgets to perform meaningful malicious tasks. In a seminal paper, Shacham [32] showed that ROP is Turing complete. Since, several real world attacks employ ROP to bypass DEP. State-of-the-art binary-level defenses against ROP approach the problem from three different paradigms: gadget elimination, control-flow integrity (CFI) and artificial diversification or randomization. Defenses based on gadget elimination statically analyze a binary and perform semantics-preserving transformation in order to eliminate gadgets [28, 27]. CFI-based defenses enforce CFI, a program property [3] that requires the flow of control during the execution of a program to adhere to a statically determined control flow graph. Due to the lack of precise program semantics, binary-level CFI defenses (e.g., BinCFI [43], CCFIR [42]) enforce an approximate control-flow graph (CFG) and as a consequence, they enforce coarse-grained CFI policies. Finally, defenses based on artificial diversification (e.g., [40], [17]) randomize at various granularities, the locations of modules in the memory thereby making it hard for an attacker to locate the gadgets.

Firstly, defenses based on gadget elimination often lack coverage due to the vast number of gadgets in binaries and the intrusiveness of the approach. For example, Pappas et al. [28] disrupt or eliminate no more than 76.9% of all the gadgets in all PE modules in Windows 7 and Windows XP. That still leaves an attacker with 23.1% (or 6,320,777) gadgets to construct an attack. Secondly, as demonstrated by Carline and Wagner [8], and Göktaş et al. [21], state-of-the-art binary-level CFI defenses (e.g., BinCFI [43], CCFIR [42]) suffer from low precision due to their coarse-grained nature. Most practical implementations of CFI including those based on shadow stack impose high performance overhead [15]. Runtime hardware-based CFI defenses ROPecker [11] and kBouncer [29] have also been shown inadequate when confronted by a determined adversary [8]. Moreover, CFI-based approaches often lack incremental deployability and offer *all-or-nothing* protection. That is, either all modules are protected or no modules are protected. Finally, artificial diversification as a solution is vulnerable to *disclosure attacks* [6, 35, 25]. By injecting and reusing the predictable just-in-time (JIT) code into a program's memory, Snow et al. [35] show that randomization is also not an effective solution. Fundamentally (also highlighted by [8]), defenses against ROP define a "gadget" to be a *short* sequence of instructions terminated by an indirect branch instruction, which is not necessarily true. In fact, [8] and [21] demonstrate attacks that utilize *large* and legitimate sequence of instructions – sometimes entire functions – as gadgets.

A key component of most ROP attacks is *stack pivoting*, a technique that positions the stack pointer to point to the ROP payload – an amalgamation of data and pointers to gadgets. In this paper, we observe that during any point in a program, the stack pointer must point to the stack region of the currently executing thread. We also observe that during ROP, each gadget behaves like an instruction with complex semantics, and the stack pointer performs

*This work was done when the author was a student at Syracuse university.

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2818000.2818023.

the role of a program counter. Therefore, within the realm of ROP, traditional CFI (i.e., integrity of instruction pointer) transforms into integrity of stack pointer. By performing compiler-level modifications during code generation, we ensure that modifications to the stack pointer lie within a predetermined region and stop attempts by an adversary to pivot the stack to point to the ROP payload that is located outside the stack region.

Our solution presents several advantages over prior binary-level ROP defenses:

1. We take a non-control-flow approach and make no assumptions regarding the size or instruction semantics of gadgets. In fact, our solution is oblivious to the concept of a "gadget", and operates at an instruction level.
2. Our solution does not depend on ASLR. Our threat model allows for ASLR to be turned off and yet, defend against ROP.
3. Our solution allows for incremental deployment. That is, only specific modules can be protected, and the protected modules can inter-operate with unprotected modules.
4. Finally, our solution defends against ROP attacks where payload is located outside the stack region (e.g., heap) with a low overhead of $\sim$1%.

Using the LLVM compiler architecture, we implemented our compiler-level solution in proof-of-concept prototypes called `PBlocker` and `PBlocker+`. `PBlocker` asserts the sanity of stack pointer whenever the stack pointer is modified, whereas `PBlocker+` asserts the sanity of stack pointer at the end of each function. `PBlocker` imposed an overhead of 1.04% for SPECINT 2006 benchmark, 1.99% for binutils and 0.7% for coreutils, whereas `PBlocker+` imposed an overhead of 2.9% on SPECINT 2006 benchmark.

The rest of the paper is organized as follows: Section 2 provides a technical background on ROP attacks and stack pivoting. Section 3 and 4 present our solution and the relevant security analysis respectively. We evaluate our solution in Section 5. We present the related work and conclude in Section 6 and 7.

2. TECHNICAL BACKGROUND AND MOTIVATION

We briefly review the various steps involved in ROP attacks. One particular step: *Stack Pivoting* is fundamental in understanding the rest of the paper.

2.1 ROP Attacks

Rerturn-Oriented Programming (ROP), an extension of return-to-libc, is a well established attack technique. During ROP attack, an attacker reuses fragments of code called "gadgets" in existing executable code regions. Traditionally, a gadget is a short sequence of instructions terminated by a `ret` instruction. By chaining multiple gadgets in the program, one can achieve meaningful computation [32]. Other variants of ROP [7], use a `pop reg` followed by an indirect `jmp reg` instruction instead of a `ret` instruction as the last instruction of the gadgets. Without loss of generality, in this work, we use the term ROP to include traditional ROP and its variants.

Instructions in x86 are of variable-width, therefore it is possible that potentially useful gadgets can be constructed by starting at an offset within an *intended* instruction. Instructions in such gadgets are termed *unintended* instructions.

A schematic overview of steps involved in a ROP attack is presented in Figure 1. Also, Figure 2 presents the concrete steps in an ROP attack. The attacker first injects the payload into the victim process' memory. Here, and in the remainder of the paper, we refer to ROP payload or payload as the combination of data and addresses of the gadgets used in the ROP attack[1]. In Figure 2, the payload resides as data at address 0x8000.

In theory, an attacker can inject ROP payload into any segment that is writable. In practice however, a vast majority of browser exploits utilize a popular and convenient technique called *Heap Spray*, wherein the payload is dumped onto the heap. Depending on the nature of the vulnerability and constraints specific to the attack, an attacker may choose to or need to inject payload in a specific writable section of the program memory.

The second step exploits the vulnerability in the victim process. This step is independent of the nature of vulnerability (e.g., use-after-free, integer overflow, buffer overflow). At the end of this step, the attacker controls the program counter. S/he may also control certain registers depending on the nature of the attack. For example, in Figure 2, a vulnerability in the victim process allows the attacker to control registers `eax` and `ebx`. The attacker achieves malicious code execution by loading `eax` with the address of the payload (0x8000) and `ebx` with address of a special type of gadget called "stack pivot".

The third step is the execution of stack-pivot gadget, which loads the address of the location where ROP payload is stored into the stack pointer. This step definitively transforms the execution to the ROP domain, and the stack pointer assumes the role of the program counter. Stack-pivoting is crucial for the attacker to convert an instance of single arbitrary code execution into continuous execution of malicious logic.

Finally, an indirect branch instructionat the end of the stack pivot gadget triggers the execution of the chain of gadgets directed by the payload (in Figure 2, $0x51c0577f$, $0x77c30083$, $0x51c05534$, etc.). Often, the scope of ROP is limited to bypassing the DEP. An executable is injected into a data page and an API such as `VirtualProtect` or `mprotect` is used to set the data page as executable. Our solution is independent of the scope/goals of ROP and therefore we do not dwell into the details of bypassing DEP.

2.2 Stack Pivoting

A requirement for stack-pivot operation is to write to the stack pointer. We refer to such instructions as "SP-update" instructions, short for "stack-pointer update" instructions. Depending on the nature of the write operation, we further classify SP-update instructions into:

- **Explicit SP-update:** These instructions perform an explicit write operation that alters the stack pointer (e.g., `mov esp, eax; add esp, 0x10;`). Explicit SP-update instructions occur in two forms:
  - *Absolute SP-update:* These instructions write an absolute value or register into the stack pointer. For example, `mov esp, eax; xchg eax, esp; pop esp;`.
  - *Relative SP-update:* These instructions alter the stack pointer by a fixed offset. For example, `add esp, 0x10; sub esp, 0x10;`.
- **Implicit SP-update:** These instructions alter the stack pointer as an implicit effect of another operation. `pop eax; ret; retn;`, etc. are examples of implicit SP-updates.

[1]ROP payload is diffent from the malicious executable payload that is commonly executed after DEP is bypassed.

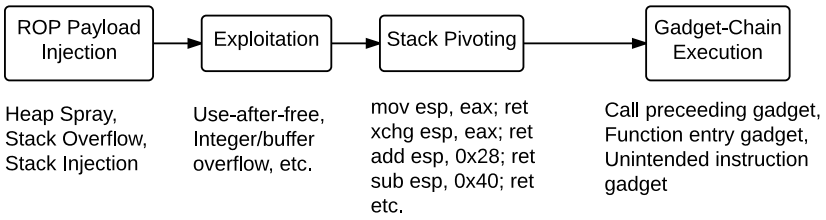


Figure 1: Steps involved in executing a typical ROP attack.

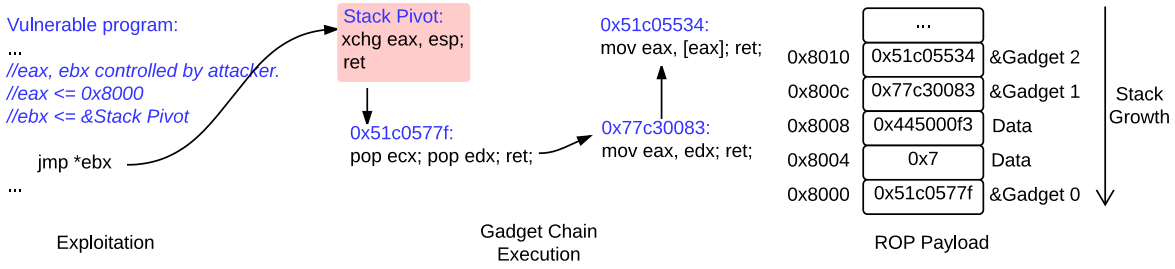


Figure 2: Example of stack pivoting

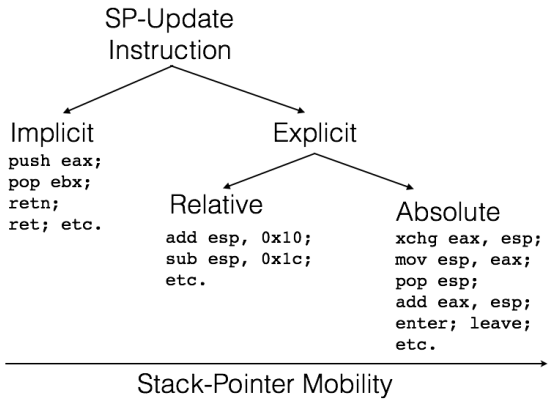


Figure 3: Classification of SP-update instructions.

Figure 3 shows the classification of various SP-update instructions. Absolute SP-update instructions can load an arbitrary value into the stack pointer, and are therefore most popular and convenient choice for pivoting. Due to the limited abilities in moving the stack pointer, relative SP-update instructions are less capable than absolute SP-update instructions and the implicit SP-update instructions are least capable. In fact, all the practical attacks we encountered accomplished pivoting using absolute SP-update instruction.

It is possible that abundance and easy availability of absolute SP-update instructions is the reason why attacks do not use relative or implicit SP-update instructions for pivoting. We believe that scarcity of absolute SP-update instructions will force attackers to construct complex attacks by using relative and implicit SP-update instructions for pivoting.

2.3 Legitimate Use Cases for Explicit SP-update Instructions

There are some legitimate use cases for explicit SP-update instructions. Under normal execution, the stack pointer of a thread is indicative of stack region being used by the thread. When a function is invoked, space on the stack – called function frame – is allocated for the function, and when the function returns, the exact amount of space that was allocated is reclaimed. Allocation and deallocation are accomplished by simply moving the stack pointer by the amount of stack space the function requires. Typically, when the size of the stack required by a function is known during compile time, the compiler inserts relative SP-update instructions to allocate and deallocate the function stack frame. For example, in LLVM clang compiler, frame allocation is accomplished via a `sub` *offset*, `%rsp` instruction or the `push` instruction, and deallocation is accomplished through `add` *offset*, `%rsp` instruction. In fact, other than frame allocation and deallocation, we found no legitimate uses of relative SP-update instructions. It is possible that `enter` and `leave` instructions are used to save and restore stack pointer and the frame pointer. These instructions manifest as absolute stack pointer updates.

Furthermore, while infrequent, the compiler sometimes introduces absolute SP-update instructions to initialize the stack pointer. When the size of a function's stack frame is unknown during compile time (e.g., when the function allocates stack space dynamically using `alloca`), the compiler inserts code to calculate the appropriate frame size at runtime and using an absolute SP-update instruction, initializes the stack pointer with the correct value. There are also legitimate uses of absolute SP-update instructions when the stack is unwound (e.g., during an exception). In such cases, the value of the stack pointer is calculated and initialized at runtime. C compilers that target flavors of Windows OS utilize a helper routine called `_chkstk` when the local variables for a function exceed 4K and 8K for 32 and 64 bit architectures respectively. The function `_chkstk` checks for stack overflow and dynamically grows the stack region using an absolute SP-update instruction if the stack growth is within the thread's allowable stack limit. More on dynamic stack allocation is presented in Section 3.4.

3. PBLOCKER

3.1 Threat Model and Scope

Our solution assumes an adversary who has the capability to exploit a vulnerability in a program and achieve arbitrary code execu-

tion. Further, irrespective of ASLR, we assume that the adversary has full knowledge of the program layout and can successfully locate useful gadgets in the memory. Strong ASLR will only improve the protection provided by our solution. We impose no restrictions on the size of the gadgets and allow an adversary to utilize large gadgets – like ones used in [21] and [8] – that can successfully evade state-of-the-art binary-level CFI defenses.

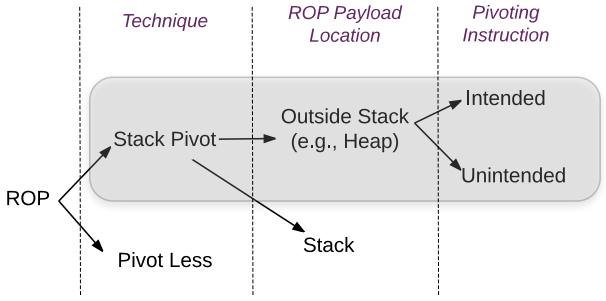


Figure 4: Scope of `PBlocker`.

Further, we assume that the attacker has injected the ROP payload into the victim memory and *requires* to perform stack pivoting in order to trigger the execution of the gadget chain. In fact, the only requirement for our solution to be a fruitful defense against ROP is that the stack pivoting be required in order to carry out the attack. A schematic representation of scope of our solution is presented in Figure 4. Our solution can protect against all ROPs that require stack pivoting and pivot outside the stack region.

Certain rare forms of ROP that do not use stack pointer as the program counter are known to exist [9, 34], and are out of our scope.

Pivoting within Stack Region.

Instances of ROP where the ROP payload is located on the stack are rare. However, with wide deployment of `PBlocker` and similar defenses, we believe that attackers can and will deploy payloads on the stack.

As shown in Figure 5(a), the attacker first injects the payload into variables in function `f2`. When `f3` is invoked, she exploits a vulnerability in `f3` and pivots the stack by adding an offset to stack pointer to point to the base of the payload in `f2`'s stack frame. Similarly, in Figure 5(b), the attacker pivots into the stale portion of the stack. First, she injects payload into `f6`'s stack frame. When `f6` returns to its caller `f5`, a vulnerability in `f5` is exploited. Finally, a pivot that subtracts an offset from the stack pointer to point to the base of the payload is executed.

Such pivots are complex to accomplish due to two challenges. Firstly, the attacker must find sufficient stack space to inject the payload. Secondly, the attacker must predict the exact location of the payload on the stack. The second challenge can be particularly hard if the location of the stack is randomized (which is often the case). For example, in the APT3 Phishing Campaign [19] attack, attackers inject a custom class with a function that accepts a large number of arguments. The arguments are placeholders for ROP payload. While we do not explore the solution to intra-stack pivoting in the current paper, we intend to pursue it in future work.

3.2 Overview

Stack Localization Property.

The execution of stack-pivot during a ROP attack signifies the transformation from regular execution to ROP. Post stack-pivot, the stack pointer assumes the role of program counter. Specifically, we observe that similar to how arbitrary code execution violates the integrity of control-flow (i.e., integrity of instruction pointer), pivoting the stack violates the integrity of stack pointer. Particularly, we define the following property that is an invariant during any point of execution.

Stack Localization (P1): At any point during execution of a program, stack frame (represented by stack pointer) of the currently executing function lies within the stack region of the currently executing thread.

Note that **P1** is true because each thread contains a dedicated stack region where the thread's stack is maintained. Under normal operation, the stack region must be in accordance with the stack allocated by the OS kernel.

Code Instrumentation.

When the payload is outside the stack region, stack-pivoting violates **P1**. We present `PBlocker` and `PBlocker+`, two implementations that enforce *Stack Localization*. Since stack pointer is indicative of the stack frame, **P1** is nothing but:

$$StackBase_{Thread} < StackPointer < StackLimit_{Thread}$$

While **P1** is an invariant and must be true at all points during the execution, it is not necessary to assert it at every point. In fact, `PBlocker` asserts **P1** only after an absolute SP-update instruction. This was sufficient to protect against all practical stack pivoting operations we looked at. After each absolute SP-update instruction, `PBlocker` retrieves the stack region allocated for the currently executing thread and asserts that stack pointer lies within the stack region. The assertion is performed using code that is instrumented through a LLVM compiler pass.

Furthermore, in order to protect against future attacks that may utilize relative and implicit SP-update instructions, we implement `PBlocker+` that performs function-level enforcement. Particularly, through instrumentation, `PBlocker+` asserts the sanity of stack pointer at the end of each function (i.e., before each `ret` instruction).

By defending against stack pivot, our solution can afford the attacker precise knowledge of gadgets in the memory. This feature distinguishes `PBlocker` and `PBlocker+` from approaches based on gadget elimination. Consider the code that is embedded into JavaScript code of some real-world exploits:

```
try {location.href='ms-help://'}catch(e){}
```

The above code loads hxds.dll, a MS Office help library that is non-relocatable and is always loaded at the same location in the memory. Moreover, it contains absolute SP-update instructions that can be used to execute a pivot. By loading hxds.dll, an attacker effectively invalidates ASLR. This is analogous to code-reuse attacks described by Snow et al. [35], but without any JIT code.

Elimination of Unintended SP-update Instructions.

As a final step, we eliminate all the unintended explicit SP-update instructions. `PBlocker` and `PBlocker+` protect against use of intended SP-update instructions as gadgets. However, an attacker can utilize the unintended instructions that could result due to misaligned instruction access. Considerable research has gone into removing unintended gadgets from the program (e.g., G-Free [27], in-place code randomization [28]). We simply leverage these efforts to render `PBlocker` or `PBlocker+` protected binaries free of unintended SP-updates.

Work-flow of our defense is presented in Figure 6. The imple-

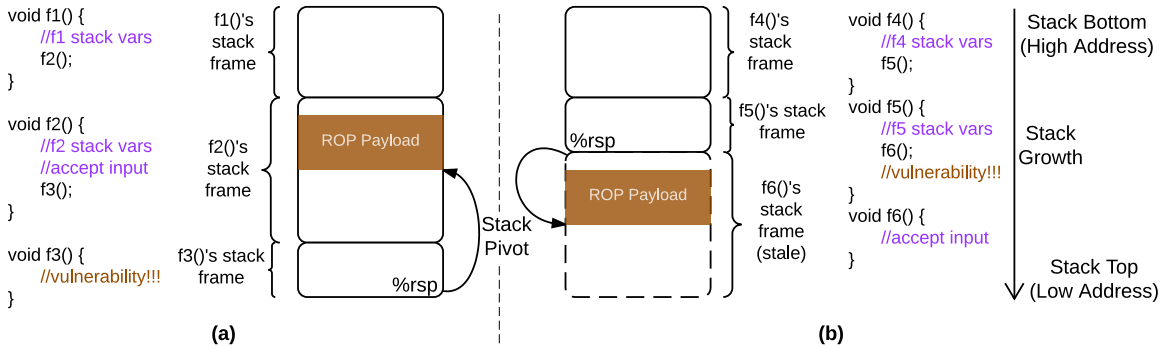


Figure 5: Pivoting within the stack region.

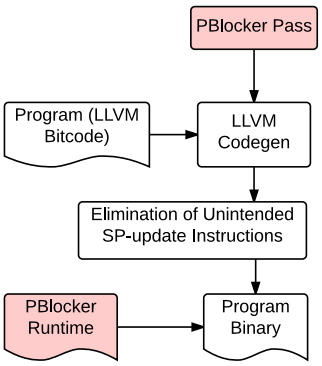


Figure 6: Work-flow of `PBlocker`. It comprises of an LLVM code generation pass that performs instrumentation, and a runtime component that provides target specific implementations.

mentation comprises of `PBlocker` Pass (or `PBlocker+` pass), a LLVM code generation pass that performs instruction-level instrumentation to capture the SP-updates, and a runtime that provides the implementation of the core functionality like assertion of **P1**. More details are provided in Sections 3.3.

3.3 Design and Implementation

The algorithm for `PBlocker` is presented in Algorithm 1. Given a $Program$, $EnforceLocalization$ iterates over each instruction in the program and identifies absolute SP-update instructions. When such an instruction is found, a call to $CoarseCheck$ is inserted *after* the instruction. Similarly, Algorithm 2 presents $EnforceLocalizationPlus$ implemented by `PBlocker+`. For each return instruction in the program, a call to $CoarseCheck$ is inserted *before* the instruction.

The goal of $CoarseCheck$ is to assert **P1** – that is, the value of stack pointer lies within the stack region of the currently executing thread. Every thread of execution has associated with it, a Thread-Specific Data (TSD) structure (Thread Information Block (TIB) in flavors of Windows OS), that contains information regarding the currently executing thread. For example, TIB contains addresses of bottom and top of stack, process ID, thread ID, exception handling related information, etc. TSD structure is directly mapped to the base of `gs` or `fs` segment registers for 64 and 32 bit variants respectively. First, $StackBase$ and $StackLimit$ of the current thread is retrieved from the TDS of the thread. If the $StackPtr$ does not lie within the interval $(StackBase, StackLimit)$, a violation of **P1** is inferred, and the execution is aborted.

$EnforceLocalization$ and $EnforceLocalizationPlus$ are implemented within LLVM `MachineFunction` passes. The implementation dependent $CoarseCheck$ is implemented within a target-dependent runtime. During the compilation, after the LLVM target code generation, for each absolute SP-update instruction, a call to $CoarseCheck$ is inserted.

Algorithm 1 Given the llvm bitcode $Program$, `PBlocker` inserts a call to $CoarseCheck$ *after* each absolute SP-Update instruction.

```
1:  procedure EnforceLocalization(Program)
2:      for each Inst in Program do
3:          if Inst is SP-Update_Absolute then
4:              Save Live Registers
5:              InsertCall CoarseCheck(StackPtr)
6:              Restore Saved Registers
7:          end if
8:      end for
9:  end procedure
10: procedure CoarseCheck(StackPtr)
11:     StackBase ← TSD.GetStackBase()
12:     StackLimit ← TSD.GetStackLimit()
13:     if StackPtr ∉ (StackBase, StackLimit) then
14:         abort()
15:     end if
16: end procedure
```

Algorithm 2 Given the llvm bitcode $Program$, `PBlocker+` inserts a call to $CoarseCheck$ *before* each `ret` instruction.

```
1:  procedure EnforceLocalizationPlus(Program)
2:      for each Inst in Program do
3:          if Inst is a return then   ▷ If this is return instruction
4:              Save Live Registers
5:              InsertCall CoarseCheck(StackPtr)
6:              Restore Saved Registers
7:          end if
8:      end for
9:  end procedure
```

'leave' Instruction.

Some compilers implement the function epilogue using `leave` – a 1 byte x86 instruction. The semantics of `leave` instruction is analogous to: `mov esp, ebp; pop ebp`. Because `mov esp,`

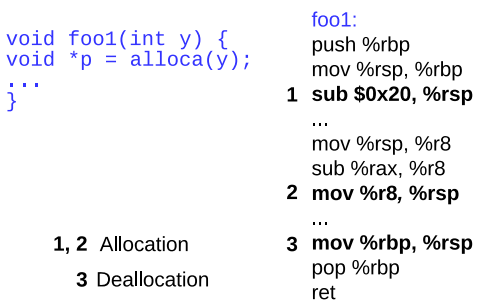


Figure 7: Dynamic allocation of stack space using `alloca`. Compiled using clang-600.0.51, based on llvm-3.5.

`ebp` is an absolute SP-update instruction, `leave` is also an absolute SP-update instruction.

It is worth noting that some compilers (e.g., clang) prefer `sub` instruction to acquire stack and `add` instruction to reclaim stack as opposed to `enter` and `leave` instructions.

3.4 Dynamic Stack Allocation

When stack space is dynamically allocated using a function like `alloca`, the user does not need to explicitly free the memory. Implementations of `alloca` are often provided by the compiler. At the time of invocation, the stack pointer is adjusted to claim the additional stack space, and when the function returns, the stack pointer is restored to its original value to account for frame deallocation. If the compiler can statically compute the requested size, it can perform the stack allocation using a relative SP-update instruction, otherwise it uses an absolute SP-update instruction.

For example, in Figure 7, in function `foo1`, the argument to `alloca` is a variable. Therefore, the compiler allocates 0x20 bytes (marking 1) required by the function, then adjusts stack pointer (%rsp) using an absolute SP-update instruction, by subtracting a value corresponding to the argument to `alloca` (marking 2). When the function completes execution, the stack pointer is simply restored to the value at function entry (marking 3). Because an absolute SP-update instruction (marking 2) is used, `PBlocker` inserts a call to $CoarseCheck$ after `mov %r8, %rsp`.

3.5 Explicit SP-update Injection through JIT

Due the unavailability of code for static-analysis based defenses, gadgets injected into Just-In Time (JIT) code by an attacker are particularly hard to protect against. However, code generator within a JIT engine can be modified to instrument all explicit SP-update instructions to enforce **P1**. Also, during code generation, unintended SP-update instructions can be avoided by using verified byte sequences for the generated code.

3.6 Interleaved Data and Code

It is possible that code and read-only data are interleaved in the executable section of a binary. While such a binary violates the fundamental tenets of DEP, unfortunately they do exist. In fact, several DLLs in Windows system directory contain read-only data interleaved within the code sections. For example, such read-only data in uxtheme.dll contain absolute SP-update instructions. Gadget elimination solutions can not eliminate gadgets in such read-only data. As a source code level implementation, `PBlocker` ensures that no data is contained within executable regions. Particularly, during the code generation phase, all data (read-only and writable) are allocated in separate non-executable sections and only executable code is allocated within the read-only executable sections.

4. SECURITY ANALYSIS

In this section we first differentiate between `PBlocker` and `PBlocker+`, and CFI. Further, as already mentioned in Section 3.1 and Figure 4, though lack of pivoting (Section 4.3) and non-stack pivots (Section 4.4) are out of our scope, we include them here to provide a deeper understanding of the problem.

4.1 PBlocker/PBlocker+ vs CFI

CFI and `PBlocker`/`PBlocker+` are fundamentally different. CFI relies on a complete (or approximately complete) control-flow graph, whereas `PBlocker` and `PBlocker+` do not. This key difference allows for `PBlocker` and `PBlocker+` to be incrementally deployable. That is, protected modules can seamlessly inter-operate with unprotected modules. This allows for protection to be applied to high-risk modules (that contain/require absolute SP-update instructions) such as mshtml.dll, uxtheme.dll, etc.

4.2 Pivoting through Implicit SP-update Instructions

In principle, implicit SP-update instructions can be used to perform stack pivoting, however they are not as powerful as the explicit SP-update instructions. Unlike explicit SP-update instructions, implicit SP-update instructions can only move the stack pointer by small increments. Considering that an attacker has just one attempt at stack pivoting after exploitation, unless the payload is close to the existing value of stack pointer, pivoting through implicit SP-updates is hard.

From the defense standpoint, there are multiple implicit SP-update instructions like `pop` *`reg`*`;`, `push` *`reg`*`;`, `ret;` that are all frequently used. Enforcing **P1** after each implicit SP-update instruction is impractical. Therefore `PBlocker+` collectively asserts **P1** before each function returns.

4.3 Stack-Pointer-Aligned Payload

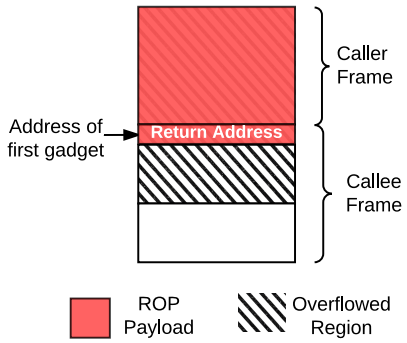


Figure 8: Stack-pointer-aligned payload.

`PBlocker` addresses the integrity of the stack pointer, which is violated during stack pivoting. While stack pivoting is required in accomplishing most real-world ROP exploits, some exceptions exist. Specifically, if an attacker can inject the payload to a location already pointed to by stack pointer, there is no need for stack pivoting. This is specially the case when the attacker can overflow the stack and control the return address (e.g., through a buffer overflow). For example, in Figure 8, through a buffer overflow in the callee function, an attacker can overwrite the return address to

point to the first gadget in the ROP payload. When the callee function returns, the ROP payload is executed.

Our solution can not protect against attacks that do not modify the stack pointer. However, buffer overflow is a well studied problem (e.g, [13, 23, 38]) with practical implementations. StackGuard [13], a popular solution incorporated into modern compilers (e.g., `-fstack-protector` in GCC and clang), introduces a randomly generated canary between the return address and the local variables of a function. When the function returns, if the canary is altered, an overflow is inferred. Most modern compilers not only include support for stack canaries, but some also incorporate them as a default setting.

4.4 Non-Stack-Pointer Pivots

The key requirement for code execution is a reliable means to repeatedly move the program counter. Under normal execution, x86 hardware increments the instruction pointer after every instruction, similarly, under traditional ROP `ret;` or `pop reg; jmp reg;` instructions allow for movement of the stack pointer that assumes the role of program counter. In principle, as long as an attacker has access to repeated indirect branching, code-reuse attacks can not be eliminated.

For example, an attacker can point a general purpose register (say `edx`) to the payload, and find an amicable gadget that repeatedly performs *update-load-branch* as discussed by Checkoway et al. [9]. Such attacks can not be defeated by `PBlocker`. However, the availability of `pop, ret` that automatically move the stack pointer would be missing in such non-stack pivots, and their practicability is unclear.

Schuster et al. introduce COOP [34], code reuse attacks for C++ programs. They leverage loops that execute virtual functions as program counter. By controlling the loop counter and the array of virtual functions that are executed, they achieve arbitrary code execution. In such code-reuse attacks, there is no need for stack pivoting.

5. EVALUATION

We implemented a prototype for `PBlocker` and `PBlocker+` as a compiler-level solution. The instrumentation phase (Figure 6) was implemented by adding a code-generation pass to the LLVM-3.5.0 compiler. As a proof-of-concept, we also implemented the target-dependent runtime for 64 bit Linux (version 3.2.0). Our LLVM pass comprises of 315 lines of C++ code for `PBlocker` and 330 lines of C++ code for `PBlocker+`. The runtime for Linux consists of 20 lines of assembly code.

5.1 Performance

We evaluate the performance of `PBlocker` and `PBlocker+` on SPECINT 2006 benchmark, and performance of `PBlocker` on GNU coreutils (ver 8.23.137) and GNU binutils (ver 2.25). The results for SPEC benchmark for `PBlocker+` and `PBlocker` are presented in Figure 10 and 9 respectively, and results for coreutils and binutils are presented in Figure 11. Overall, we found that `PBlocker` and `PBlocker+` impose very little overhead. Average overhead of `PBlocker` was found to be 1.04% for SPEC benchmark, 1.99% for binutils and 0.7% for coreutils. This is due to the infrequent use of absolute SP-update instructions in the binary. For example, 5 out of 9 programs that we tested in coreutils contained no absolute SP-update instructions.

Furthermore, `PBlocker+`, which is a more conservative and strict defense imposed 2.9% overhead for the SPECINT benchmark.

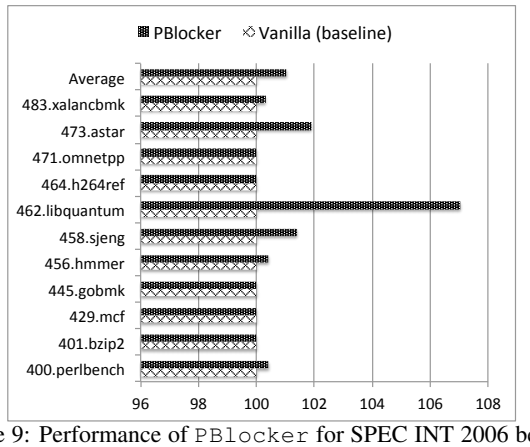


Figure 9: Performance of `PBlocker` for SPEC INT 2006 benchmark normalized against vanilla LLVM-3.5.0. The x-axis is adjusted in order to clearly indicate the overhead.

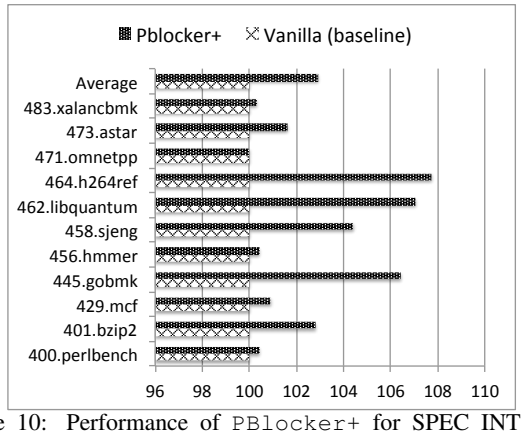


Figure 10: Performance of `PBlocker+` for SPEC INT 2006 benchmark normalized against vanilla LLVM-3.5.0. Policy is enforced before each function returns. The x-axis is adjusted in order to clearly indicate the overhead.

5.2 Pivoting in Practice

In Table 1, we present some modules in Windows OS and the common absolute SP-update instructions within them. We found `xchg eax, esp;` to be the most common pivoting instruction. Also, in Table 3, we present a corpus of recent exploits on Metasploits [1] and the instructions they utilize to accomplish pivoting. Unsurprisingly, they use the `xchg eax, esp;` instruction. It must be noted that exploits on Metasploit are proof-of-vulnerability, and pivoting is independent of the vulnerability. That is, depending on the attack specifics, a feasible pivot can be utilized for multiple exploits. However, in practice absolute SP-update instructions are most popular to perform stack pivot.

Moreover, hxds.dll – the help library for MS Office is not relocatable and always loads at the same address. An attacker can simply load and utilize the pivot gadgets within the module. `PBlocker` and `PBlocker+` are particularly useful in protecting such non-relocatable modules. `PBlocker` can defeat pivoting in all cases listed in Table 1 except the gadget at uxtheme.dll:0x6ce8ab5e because uxtheme.dll contains readonly data interleaved with code in the `.text` segment, and data item `char s_keyPublic1[]` is at address 0x6ce8ab38.

5.3 SP-Update Instructions vs Gadgets

Table 1: Absolute gadgets in Windows OS.

Program	Gadget Module	Gadget Address	Pivot Instruction	Relocatable	PBlocker defeats pivot?
Office 2007	hxds.dll	0x51c2213f	`xchg eax, esp`	NO	✓
Office 2010	hxds.dll	0x51c00e64	`xchg eax, esp`	NO	✓
Win XP SP3	msvcrt.dll	0x77C3868A	`xchg eax, esp`	Yes	✓
Java Runtime	NPJPI.dll	0x7c342643	`xchg eax, esp`	Yes	✓
Apple QT	QickTime.qts	0x20302020	`pop esp`	Yes	✓
Adobe Flash	flashplayer.exe v11.3.300.257	0x1001d891	`xchg eax, esp`	Yes	✓
Win 7	uxtheme.dll	0x6ce7c905	`mov esp, ebp`	Yes	✓
Win 7	uxtheme.dll	0x6ce8ab5e	`mov esp, [edi + 0xffffffcd]`	Yes	X

Table 2: Explicit SP-Update instructions vs Gadgets

Suite	Program	Total Instructions	# Absolute SP-update	# Relative SP-update	Total # Gadgets
coreutils	rm	9470	0	117	705
	cp	17403	14	170	985
	factor	9907	4	118	890
	sha512	9969	0	77	547
	sort	19471	0	158	1053
	cat	6704	4	133	475
	wc	5400	0	77	490
	md5sum	5659	0	71	441
	split	9888	4	108	579
binutils	objdump	265075	49	1524	11673
	objcopy	230226	16	1366	9921
	ld	48964	1	705	2860
	nm	189299	16	1104	8604
	ar	192428	16	1118	8936
	readelf	60170	31	207	3868

Table 3: Pivoting instructions used by recent Metasploit exploits

CVE Number	Instruction
2013-3897 2013-3163 2013-1347 2012-4969 2012-4792 2012-1889 2012-1535	`xchg eax, esp`
2014-0515	`mov esp, [eax]`
2013-1017	`pop esp`

In order to demonstrate the effectiveness of PBlocker, we list the number of explicit SP-update instructions that PBlocker protects as opposed to the total number of gadgets in coreutils and binutils. The results are tabulated in Table 2. Overall, we found that absolute SP-update instructions, the most popular for stack-pivoting are a very small fraction when compared to the total instructions in a program.

6. RELATED WORK

6.1 Defense against Stack Pivoting

Microsoft's Enhanced Mitigation Experience Toolkit (EMET) [2] is a popular zero-day exploit prevention utility that provides defense against stack pivot in ROP attacks. Core idea implemented by EMET is based on ROPguard [20]. When execution enters a critical function such as VirtualProtect, EMET asserts that stack pointer lies within the stack region of the current thread. DeMott [18] bypass EMET by taking advantage of the gap between *time-of-check* and *time-of-use* of stack pointer. They first perform a stack pivot to the heap, perform ROP, and then pivot back to the stack region just before invoking Virtual Protect. Because EMET checks for sanity of stack pointer within the critical function, such an attack is missed.

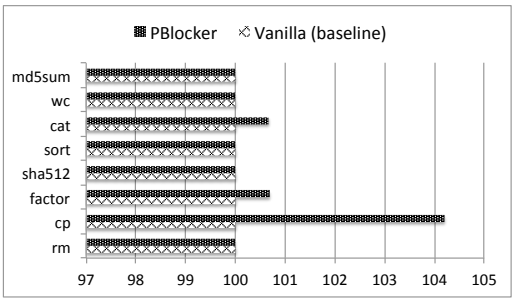
PBlocker checks the sanity of stack pointer *immediately after* every absolute SP-update instruction and can therefore stop attacks demonstrated by DeMott.

Recent stack-based defenses make stack pivoting harder, but not impossible to execute. In StackArmor [10], Chen et al., randomize the location of the stack, thereby making it harder for an attacker to guess the location of ROP payload on the stack. However, they are still vulnerable pivoting, if an attacker can successfully locate the payload on the stack [19].

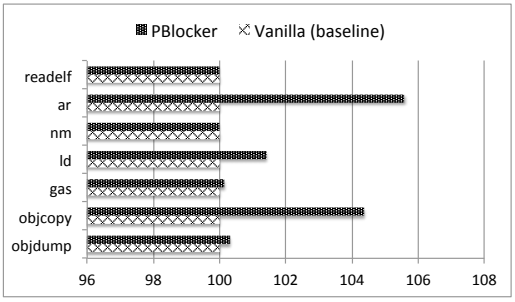
6.2 CFI

Beginning from Abadi et al. [3], several CFI defenses have been proposed on the source code [22, 37], at a binary level [43, 42, 16, 14] and runtime [39, 31]. While different from CFI, Kuznetsov et al. propose CPI [24], which distinguish between code and non-code pointers and protects code pointers. Unlike CFI, PBlocker and PBlocker+ are not control-flow-based approaches, however they are complementary to CFI. Coarse-grained CFI can supplement PBlocker and PBlocker+ to prevent an attacker from utilizing unintended SP-update instructions (particularly 1 byte instructions like leave) that can not be removed using gadget elimination techniques.

More recent CFI defenses on the binary like, vfGuard [30] improves the precision of CFI for C++ virtual call dispatches, and



(a) Sampling of coreutils.



(b) Sampling of binutils.

Figure 11: Performance of `PBlocker` for a few coreutils and binutils programs normalized against vanilla LLVM-3.5.0. The x-axis is adjusted in order to clearly indicate the overhead.

Opaque CFI [26] combines coarse-grained CFI and artificial diversification in order to render disclosure attacks harder.

6.3 Artificial Diversity

The goal of artificial diversity is to randomize and hide the location of a program's code, data, stack, heap, etc. [4, 36, 5, 41, 12]. STIR [40] performs static instrumentation to generate binaries that self-randomize every time the binary is loaded. Isomeron [17] combines code randomization with execution-path randomization wherein code fragments that can be indirectly targeted are duplicated, and at runtime, a randomly chosen fragment from the duplicates is invoked. Xu and Chapin [41] introduce ASLR using code-islands in order to defend against chained return-to-libc attacks, wherein they identify and randomize into isolated code blocks, base pointers used in memory mappings.

Artificial-diversity-based defenses are susceptible to disclosure attacks, and are not always an effective defense [33]. `PBlocker` and `PBlocker+` do not rely on ASLR for defense.

6.4 Gadget Elimination

Two main works: in-place code randomization [28], G-Free [27] have been proposed to eliminate gadgets. Given the vast number of available gadgets even in binaries [32], it is hard to eliminate *all* the gadgets in a program. They perform semantics-preserving in-place code randomization.

7. CONCLUSION

In this paper, we presented `PBlocker`, a novel defense against ROP attacks. `PBlocker` enforces *Stack Localization* to defend against ROP by stopping stack-pivot operations that pivot outside the stack region. This covers most of the cases of ROP. We also present `PBlocker+`, a more conservative version of `PBlocker` wherein, the stack pointer is checked before each function returns. We evaluate `PBlocker` on SPEC 2006 benchmark and show an average runtime overhead of under 1.04%.

8. ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their feedback. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-15-2-0106, and DARPA CGC Grant #FA8750-14-C-0118. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] Metasploit penetration testing framework. http://http://www.metasploit.com/.

[2] Microsoft Enhanced Mitigation Experience Toolkit. http://support.microsoft.com/kb/2458544, August 2014.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 340–353, 2005.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security*, volume 3, pages 105–120, 2003.

[5] S. Bhatkar and R. Sekar. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *IEEE Symposium on Security and Privacy (SP'2014)*, pages 227–242. IEEE, 2014.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[8] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security'14)*, 2014.

[9] 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*, pages 559–572. ACM, 2010.

[10] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

[11] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[12] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, 2002.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[14] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)*, 2014.

[15] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, volume 15, 2015.

[16] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-r. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[17] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Symposium on Network and Distributed System Security (NDSS'15)*.

[18] J. DeMott. Bypassing EMET 4.1. https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf.

[19] E. Eng and D. Caselden. Operation Clandestine Wolf – Adobe Flash Zero-Day in APT3 Phishing Campaign. https://www.fireeye.com/blog/threat-research/2015/06/operation-clandestine-wolf-adobe-flash-zero-day.html.

[20] I. Fratric. Runtime prevention of return-oriented programming attacks. http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf, 2014.

[21] 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.

[22] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] W. Lian, H. Shacham, and S. Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[26] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming using in-place Code Randomization. In *IEEE Symposium on Security and Privacy (SP'2012)*, pages 601–615, 2012.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security*, 2013.

[30] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[31] 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 (ASIACCS'13)*, pages 311–322, 2013.

[32] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[33] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[34] F. Shuster, T. Tendyck, C. Liebchen, L. Davi, A.-r. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming, On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of 36th IEEE Symposium on Security and Privacy (Oakland'15)*, 2015.

[35] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (SP'2013)*, pages 574–588, 2013.

[36] P. team. PaX: Address space alyout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[37] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*, pages 941–955, 2014.

[38] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*, pages 2000–02, 2000.

[39] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (Oakland'10)*, pages 380–395, 2010.

[40] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)*, pages 157–168. ACM, 2012.

[41] H. Xu and S. J. Chapin. Address-space layout randomization using code islands. *Journal of Computer Security*, 17(3):331–362, 2009.

[42] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'13)*, pages 559–573, 2013.

[43] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)*, pages 337–352, 2013.