Supplementing Modern Software Defenses with Stack-Pointer Sanity

Anh Quach, Matthew Cole, Aravind Prakash
Binghamton University
{aquach1,mcole8,aprakash}@binghamton.edu

ABSTRACT

The perpetual cat-and-mouse game between attackers and software defenders has highlighted the need for strong and robust security. With performance as a key concern, most modern defenses focus on control-flow integrity (CFI), a program property that requires runtime execution of a program to adhere to a statically determined control-flow graph (CFG). Despite its success in preventing traditional return-oriented programming (ROP), CFI is known to be ineffective against modern attacks that adhere to a statically recovered CFG (e.g., COOP).

This paper introduces stack-pointer integrity (SPI) *as a means to supplement CFI and other modern defense techniques*. Due to its ability to influence indirect control targets, stack pointer is a key artifact in attacks. We define SPI as a property comprising of two key sub-properties - *Stack Localization* and *Stack Conservation* - and implement a LLVM-based compiler prototype codenamed SPIglass that enforces SPI. We demonstrate a low implementation overhead and incremental deployability, two of the most desirable features for practical deployment. Our performance experiments show that the overhead of our defense is low in practice. We opensource SPIglass for the benefit of the community.

ACM Reference Format:
Anh Quach, Matthew Cole, Aravind Prakash. 2017. Supplementing Modern Software Defenses with Stack-Pointer Sanity. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3134600.3134641

1 INTRODUCTION

Software attacks that compromise and hijack control-flow continue to be a real and existential threat. Widely deployed hardware mechanisms that prevent data execution (DEP/NX) have given rise to modern "code-reuse" attacks, i.e., attacks that reuse existing code to accomplish malice [35]. Return-oriented programming (ROP) is an example of code-reuse attack wherein an attacker executes a carefully selected sequence of "gadgets". A gadget is a short sequence of instructions terminated by an indirect branch instruction (e.g., ret). A popular form of defense against modern attacks involves embedding software with security primitives in an effort to harden programs and render them attack resilient. Enforcing program integrity properties, especially control-flow integrity (CFI [3]) through inlined reference monitors is a well-studied and established defense technique. CFI requires the runtime execution of a program to adhere to a statically determined control-flow graph (CFG). CFI-based solutions that operate on both source code [3, 39] and binary [17, 31, 44, 45] have been proposed.

Practical CFI-based defenses are known to suffer from the following limitations: (1) Lack of a precise CFG: indirect branches in a program make full CFG determination hard, if not impossible [34, 41]. Without a complete CFG, defenses compute an approximate CFG containing redundant forward and reverse edges [45]. These redundancies result in attack space. (2) Lack of temporal sensitivity: while the CFG captures *what* control transitions are legal, it does not reflect *when* those transitions are legal. These limitations are exasperated by practical considerations like support for incremental deployment, support for dynamically loaded modules, etc. (3) Data hiding problem: meta-data based solutions including the shadow stack solutions depend on integrity of the shadow stack. That is, their success depends on effective hiding of data in the memory. However, hiding data in user space is hard [27] and an open research problem. Other approaches for shadow stack either require special hardware or incur significant performance overhead.

By abusing the limitations of practical CFI, recent attacks including Counterfeit Object-Oriented Programming (COOP [20, 36]) and Printf-Oriented Programming [8] demonstrate that even sophisticated defenses can be successfully evaded. While more recent defensive efforts have focused on improving the performance of practical CFI, Carlini et al. [8] show that *even fully-precise static CFI solutions can be defeated*. In essence, a solution with strong and robust security is more useful than well-performing solution lacking in security.

In this paper, we approach defense from a fundamentally different, yet complimentary point of view when compared to traditional control-flow-based defenses. We note that attacks often abuse stack and violate its intended use. Particularly, *attacks violate the integrity of stack pointer*, so we focus our defense around sanity of stack pointer. A key difference between our approach and CFI is that we do not seek to monitor the control flow of the program, whereas we seek to enforce legal use of stack pointer. By doing so, our solution can not only operate as a standalone solution, but also cooperate with existing CFI-based defenses, and enhance the overall security.

We define Stack-Pointer Integrity (SPI) as a program property that captures the intended movement of stack pointer, and design a system that enforces this property. Shifting perspective from

ACSAC 2017, December 4–8, 2017, Orlando, FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134641

instruction-pointer-centric defense (CFI) to stack-pointer-centric defense (SPI) presents multiple advantages.

(1) Unlike flow of control, flow of stack pointer is well defined, and we are therefore not constrained by completeness and precision of the CFG.
(2) Rules that govern the movement of stack pointer are independent of the location of data/code in the memory. As such, SPI does not depend on address-space layout randomization (ASLR).
(3) Unlike solutions that focus on integrity of stack content [9], SPI focuses on integrity of the stack pointer. By doing so, we lessen the performance penalties incurred in validating the stack contents.
(4) Finally, SPI and CFI are orthogonal solutions, and can complement each other to provide strong defense.

We develop the SPI policy for programs executing on the x86 architecture and implement a prototype system codenamed SPIglass. SPIglass is an LLVM-based compiler that instruments programs with SPI checks that are performed during runtime. We also provide techniques to enable interoperability between protected and unprotected modules.

Can SPI defeat *all* code-reuse attacks? Although SPIglass can thwart several practical code-reuse attacks on its own, it is designed to supplement practical CFI implementations to provide stronger security. This is particularly important given that most practical CFI efforts have been defeated [8].

Our contributions can be summarized as follows:

(1) We define Stack-Pointer Integrity (SPI) as a program property that complements CFI and comprises of two sub-properties *Stack Localization* and *Stack Conservation*.
(2) We implement SPIglass, a LLVM-based prototype that implements SPI. SPIglass can defeat modern code-reuse attacks, including COOP.
(3) We evaluate SPIglass, and show low execution overhead and reasonable memory overhead.
(4) We opensource our implementation of SPIglass to aid in future research and development[1].

2 TECHNICAL BACKGROUND

During a code-reuse attack, the attacker repeatedly executes the *ULB* sequence: *Update* the processor state, *Load* the address of next gadget into instruction pointer, and *Branch* to it. Depending on whether or not the stack pointer is used in ULB sequence, code-reuse attacks can be divided into two categories.

2.1 Attacks that Use Stack Pointer for ULB Sequence

Due to the availability of multiple instructions (e.g., `push, pop`) that update the stack pointer and the ready availability of `ret` instruction that loads and branches to the address at the top of stack, stack pointer is widely used in ULB sequences. These are the most common ROP attacks wherein the stack pointer assumes the role of instruction pointer. An example of such an attack can be found in Figure 1. First, an attacker injects a payload that contains interleaved code pointers and data. Each pointer points to a gadget, and the sequence of gadgets together accomplish malicious computation. Next, the attacker exploits the vulnerability, and obtains control of the instruction pointer. The attacker utilizes a special gadget (Gadget 0 in Figure 1) to position the stack pointer to the base of the injected payload, i.e., RSP_2 to RSP_3. This step, called stack pivoting, transitions execution into the ROP domain. The gadget used to pivot is called a pivot gadget. Finally, the `ret` instruction in the pivot gadget triggers the execution of the remaining gadgets in the payload. In Figure 1, the control flow is represented as a solid arrow whereas the movement of stack pointer is represented as a dashed arrow.

Stack pivoting. Stack pivoting positions the stack pointer to the base of the injected payload. Instructions that alter the stack pointer are called 'SP-Update' instructions [32]. They are divided into explicit SP-update instructions or instructions that explicitly alter the stack pointer (e.g., `mov rax, rsp; add rsp, 0x10;`), and implicit SP-update instructions that alter the stack pointer as an implicit effect of another operation (e.g., `pop rax; ret;`). Explicit SP-update instructions are further divided into absolute (e.g., `mov rax, rsp;`) and relative (e.g., `add rsp, 0x10`) SP-update instructions.

A stack pivot gadget comprises of an SP-update instruction followed by an indirect branch instruction. Depending on the location of payload, one of two forms of pivoting can occur:

(1) *Inter-segment pivoting:* the payload is located in a writable segment other than the stack (usually heap).
(2) *Intra-stack pivoting:* the payload is located on the stack segment. Typically, the attacker injects payload into the local variables/arrays of a function. For example, in Figure 3, the array in F6 is used to store user input, which contains the payload. Here, pivot gadget displaces the stack pointer within the stack segment.

Mobility of the stack pointer depends on the specific SP-update instruction in the gadget. Absolute SP-update instructions offer the attacker with an ability to initialize the stack pointer with an arbitrary value, and are therefore most favorable for pivoting. Relative SP-update instructions move the stack pointer by fixed offsets, and therefore offer moderate mobility. Implicit SP-update instructions move the stack pointer by small offsets, offer least mobility, and are least useful for pivoting.

2.2 Attacks that Do Not Use Stack Pointer for ULB Sequence

While stack pointer is commonly used in ULB sequence, it can be abused in other ways. COOP [36] is an exemplar of code-reuse attack without stack pivoting. It is a C++-based attack that takes advantage of the dynamic dispatch mechanism in C++, and reuses entire virtual functions as gadgets. Virtual functions in C++ are dispatched using VTable—a per-object table that contains all the polymorphic functions an object can invoke. COOP reuses a special type of gadget called main-loop gadget that executes a single virtual function on an array of objects. By injecting carefully ordered and potentially overlapping array of fake objects that point to

[1]The source code is available at https://github.com/bingseclab/spiglass

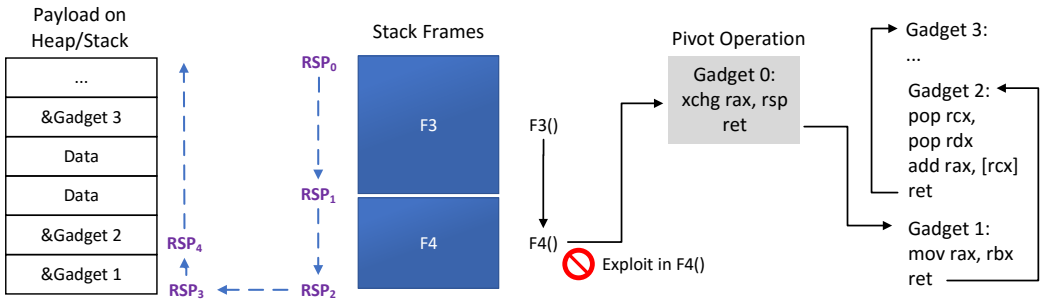

Figure 1: Movement of stack pointer in attacks that use stack pointer for ULB sequence. Stack grows downwards.

fake VTables in the victim memory, COOP achieves arbitrary code execution. We refer the readers to [36] for more details.

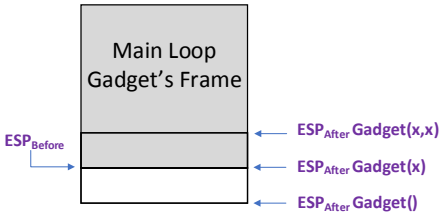

Figure 2: Stack realignment in COOP (32 bit). In this example, the virtual function that is called in the main-loop gadget accepts one argument. ESP_{Before} and ESP_{After} represent values of stack pointer before and after the execution of gadgets. Stack grows downwards.

Consider Figure 2, a mismatch in the number of arguments accepted by the gadgets versus number of arguments supplied at the invocation site in the main-loop gadget will result in a misalignment of stack pointer before and after the call to gadget (refer Figure 7 in [36] for details). This is particularly a problem in 32-bit environment where all the arguments are passed on the stack.

The values of the stack pointer before (ESP_{Before}) and after (ESP_{After}) invocation of gadgets with 0, 1 and 2 arguments are shown in Figure 2. In order to prevent stack corruption, the attack must either restrict the gadgets to only those functions that accept the same number of arguments that are supplied at the invocation site, or accommodate compensatory gadgets that realign the stack pointer after each misalignment—e.g., gadget that accepts 2 arguments followed by gadget that accepts no arguments. The former choice greatly reduces the number of gadgets available to accomplish the attack, therefore COOP uses the latter technique to align the stack pointer.

2.3 Stack Pointer Updates in Benign Execution

Below, we list common operations that alter the stack pointer during benign execution, and the instructions used to accomplish them.

Control-transition using call-ret. Implicit SP-update instructions `call` and `ret`, and their variants (e.g., `retn`) automatically store and retrieve return address from the stack, and therefore update the stack pointer.

Save/Restore registers. Depending on the calling convention, certain registers (`rbx`, `rbp`, `r12-r15` on Linux x86-64) are saved on the stack by the callee and restored before returning to the caller. The `push` and `pop` instructions are typically used to save/restore registers. Registers are saved before allocation of a function frame and restored after deallocation of the frame.

Frame allocation and deallocation. Local variables are allocated within the function frame on the stack. Allocation and deallocation are typically achieved using relative SP-update instructions by moving the stack pointer by a fixed offset along and against the direction of stack growth respectively (e.g., `sub rsp, 0x10; add rsp, 0x10`).

Dynamic frame allocation and deallocation. Absolute SP-update instructions are used to restore the stack pointer with a previously saved value when the size of a stack frame is not known during compilation, or during irregular flows like `longjmp`, exceptions, etc. These are rarely encountered.

3 THREAT MODEL AND ASSUMPTIONS

We accommodate a highly capable attacker with full knowledge of the loaded modules and the gadgets within—with or without ASLR enabled. We assume that the attacker is able to generate a payload comprising of potentially large call-preceded and function-entry gadgets such that known defenses (e.g., [44, 45]) can be bypassed. In a call-preceded gadget, the gadget is preceded by a `call` instruction and forms a valid backward-edge for return instructions in the CFG. A function-entry gadget begins at the entry point to a function and is a valid forward-edge for indirect call instructions [18]. Further, the attacker is able to inject payload into the vulnerable program, and has the capability to exploit a vulnerability in a program and achieve arbitrary code execution. Because the payload may be injected on the stack, modern defenses against stack-pivoting [32] can be successfully bypassed. We also assume that the attacker is unable to execute unintended instructions (i.e., instructions obtained by offsetting into legitimate instructions). Finally, all gadgets - including a potential pivoting gadget - are not only sequence of intended instructions, but also call-preceded or function-entry gadgets. Note that multiple efforts that focus on elimination of

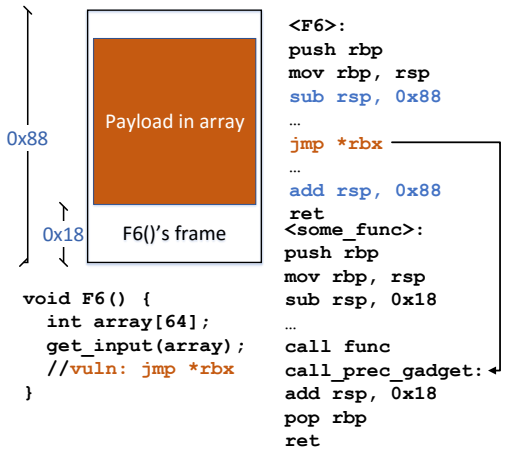

Figure 3: Pivoting within the stack region. A call-preceded gadget in `some_func` is abused as a pivot gadget through a vulnerability in function `F6`.

unintended gadgets [28, 29] and the state-of-the-art in CFI make similar assumptions [44, 45].

Consider Figure 3 where the attacker injects payload into a function's stack frame and exploits a vulnerability to invoke the call-preceded gadget in another function to pivot the stack pointer. These attacks can manifest as payload injection in one frame and exploit in another frame, and are known to be practical [15]. While pivoting-based solutions [14, 32] can not defend against such attacks, they are well within the scope of our solution.

4 SPI – OVERVIEW

4.1 SPI – Property

This work is driven by the observation that *code-reuse attacks violate the intended use of the stack pointer.* In attacks that use stack pointer in the ULB sequence, the stack pointer is abused to assume the role of an instruction pointer, whereas in the case of modern COOP-type attacks, the intended use of stack pointer is violated and results in mis-alignment in stack frames.

We define the integrity of stack pointer using properties that capture its normal behavior. We define SPI using two sub-properties:

- **P1 Stack Localization:** The stack pointer always resides within the stack region allocated for the current context (i.e., thread) of execution. That is:
  $StackBase_{Thread} < StackPointer < StackLimit_{Thread}$
- **P2 Stack Conservation:** The stack pointer is conserved across function invocation. That is:
  $StackPointer_{Before\_Func} == StackPointer_{After\_Func}$

4.2 Our Approach

Property **P1** has already been enforced by PBlocker [32] with low overhead. This paper extends PBlocker to prevent not only inter-segment pivoting, but also intra-stack pivoting. We reuse the techniques implemented by PBlocker, and focus our effort on effective enforcement of **P2**. We pursue the following enforcement goals:

Performance: Verifying the stack pointer values before and after a function execution by utilizing a shadow stack is a potential solution to enforce **P2**. Shadow-stack based solutions are well studied [12] in the context of CFI. However, the shadow stack must be protected, and updates to the shadow stack must be monitored in order to prevent corruption. Hiding a shadow stack in user space is most performance-friendly, yet hiding data in user space is hard [27]. If the attacker can locate the shadow stack, she or he may be able to corrupt it and evade the security mechanism.

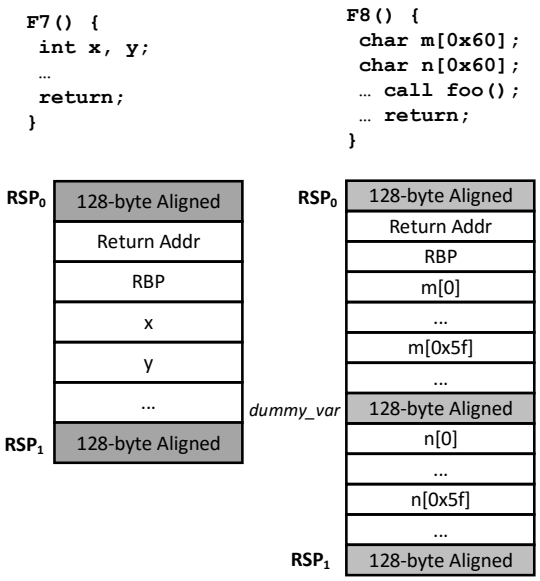

Figure 4: Function frames generated by SPIglass for F7() and F8(), small and large function frames respectively. A dummy variable is inserted in order to prevent allocation of a local variable at the *n*-byte boundary.

We make a memory-to-security trade off and improve security at the cost of memory. Cost of memory has steadily reduced over the decades. *Memory overhead is more tolerable than runtime performance overhead imposed by a security solution.* Because the stack pointer is indicative of the function frame allocation and deallocation, movement of stack pointer can be greatly controlled by adjusting the sizes of function stack frames. Specifically, if the size of each function stack frame were *n*-byte aligned, then legal stack pointer values must also be *n*-byte aligned. Using compiler modifications, we force the stack frame of each function to be *n-byte* aligned by sanitizing the stack pointer after each explicit SP-update instruction in the module. Further, we reserve *n*-byte aligned locations in functions with frames larger than *n* bytes by inserting a dummy variable. This ensures that no legal variables are allocated at aligned addresses. We sanitize the dummy variables to disrupt possible attacks that overflow into the dummy variable. Stack frames and the instrumented code for sample functions F7 and F8 are shown in Figures 4 and 5 respectively. Our solution imposes memory overhead that is approximately linear to the runtime depth of the callstack. More details can be found in Section 6.

```
<F7>:
push rbp
mov rbp, rsp
sub rsp, 0x10
…
add rsp, 0x10
pop rbp
ret
```

```
<F7_protected>:
push rbp
mov rbp, rsp
sub rsp, (n-2*DWORD_SIZE)
and rsp, MASK(n)
…
and rsp, MASK(n)
add rsp, (n-2*DWORD_SIZE)
pop rbp
add rsp, (DWORD_SIZE)
and rsp, MASK(n)
sub rsp, (DWORD_SIZE)
ret
```

```
<F8>:
push rbp
mov rbp, rsp
sub rsp, 0xD0
…
call/jmp loc
…
add rsp, 0xD0
pop rbp
ret
```

```
<F8_protected>:
push rbp
mov rbp, rsp
sub rsp, (2n-2*DWORD_SIZE)
and rsp, MASK(n)
…
mov dummy, $0x0
call/jmp loc
…
mov dummy, $0x0
and rsp, MASK(n)
add rsp, (2n-2*DWORD_SIZE)
pop rbp
add rsp, (DWORD_SIZE)
and rsp, MASK(n)
sub rsp, (DWORD_SIZE)
ret
```

Figure 5: Assembly code for unprotected and protected function F7. The stack allocation (sub) and deallocation (add) are modified to adjust allocation to alignment boundaries, and the stack pointer is sanitized in the function prologue and epilogue. A dummy variable is inserted in F8, and is sanitized before any control transition. An adjustment of two DWORD_SIZE is made in order to accommodate the saved RBP register and the return address.

Interoperability: Support for interoperability means the ability for protected and unprotected modules to seamlessly invoke functionality within each other. In order to support interoperability between protected and unprotected modules, we detect entry points within the protected module and generate wrappers that (1) save the stack pointer, (2) align the stack pointer to n byte boundary, (3) invoke the functionality in the protected module, and (4) restore the stack pointer before returning control to the unprotected module.

Backward Compatibility: We provide an implementation of SPI in SPIglass, an LLVM-based prototype. SPIglass generates programs that can execute without any modifications to the hardware or the operating system.

5 STACK-POINTER INTEGRITY (SPI)

5.1 Defense Policy

SPIglass reserves each n-byte aligned address for the stack pointer. Therefore, the function frame allocation and deallocation occur in multiples of n-byte frames where n is a power of 2. This allows low-overhead sanity checks on stack pointer.

Stack pointer invariant. After the allocation and deallocation of each function frame, the following invariants hold:

I1 Within a function body, the stack pointer will be n-byte aligned.

I2 No local variable will occupy an n-byte aligned memory location.

I2 is essential to ensure that an attacker does not inject payload into an n-byte aligned local variable, and then pivot to that location using a gadget. Further, by setting n to be a power of 2, SPIglass can quickly sanitize the stack pointer using a single instruction, which greatly reduces enforcement overhead. A detailed evaluation of performance with respect to alignment is presented in Section 6.

During compilation, the compiler sets up a stack frame to accommodate local variables in a function. SPIglass enlarges each stack frame to occupy one or more n-byte aligned frames. This ensures that the stack pointer will be n-byte aligned at all times except during function prologue and epilogue (i.e., stack allocation and deallocation). Further, because the enlarged frame extends beyond already allocated variables, relative offsets of variables within a frame remain unchanged. Code to sanitize the stack pointer is inserted after each allocation and before each deallocation of stack frame to assert n-byte alignment.

5.2 Function Frames Smaller than n bytes

If the stack space required by a function is less than the alignment size, SPIglass assigns a single aligned stack frame for the function. Furthermore, appropriate adjustments (a multiple of DWORD_SIZE) is added to accommodate the storage of return address and callee saved registers on the stack. This results in a memory overhead which is evaluated in Section6.

5.3 Large Function frames

If the stack size required by the function is greater than the alignment size, it is possible for a local variable to be allocated on n-byte alignment, which violates **I2**. In such cases, SPIglass partitions the function's stack frame into multiple n-byte aligned frames.

Frame partitioning. An algorithm to partition large stack frames is presented in Algorithm 1. It is a simplified version of the knapsack algorithm. Each partition contains local variables of the function and a dummy variable in each n-byte aligned address. Procedure `InsertDummyVars` allocates variables into frames until the n-byte boundary is reached. Then, a new aligned frame is allocated and a dummy variable is inserted at the boundary. The process is repeated until all the variables in the function are allocated.

Although the dummy variable is not directly referenced in the code, it might be possible for an attacker to overflow one of the local variables to inject a payload beginning at the dummy variable. In order to prevent such attacks, SPIglass sanitizes all the dummy variables in a function before any direct or indirect branch instruction (line 30 in Algorithm 1). Because the number of functions with large frames is relatively low, the overhead incurred in sanitizing dummy variables is low.

Furthermore, if a variable of a function (e.g., a large array or a structure) is larger than the aligned frame size, overlap on to subsequent frames can not be avoided. Such functions can therefore not be protected. In Section 6, we provide a distribution of stack frame size, and make recommendations for optimal alignment.

5.4 Interoperability

While the unprotected modules do expose an attack surface, and may be used by an attacker for pivoting, interoperability between protected and unprotected modules is essential for practical use. Firstly, it allows incremental deployment of the defense where some modules are protected and some or not. Second, it allows enforcement strategies where a trade-off between risk and memory overhead can be made. For example, protections can only be applied to those modules that show high occurrences of explicit SP-update instructions. We identify two interaction scenarios between modules:

Protected to Unprotected (P2U): this control transition occurs when a protected module invokes functionality in an unprotected module. For example, if libc is not protected and a protected module invokes functions in libc, the control transitions from a protected module to an unprotected module (e.g., calls to printf, malloc). P2U is inherently supported by SPIglass. Because the stack pointer is not sanitized in the unprotected module, and the stack frame is conserved when the control returns to the protected module, the sanity checks in the protected module are unaffected by the call and return from the unprotected module.

Unprotected to Protected (U2P): this control transition occurs when an unprotected module invokes functionality in the protected module. Because the unprotected module does not align the stack pointer, the sanity checks in the protected module will corrupt the stack pointer before returning control to the unprotected module. In order to prevent stack pointer corruption, SPIglass intercepts the control at *all* the entry points in the protected module, and aligns the stack pointers to the *n*-byte boundary before resuming the execution at the entry point.

Entry points in the protected module. We identify four types of entry points to a module:

(1) Entry point function: This is the `main` function of an executable or the `init` function of a library.
(2) Exported functions: Exported functions are explicit entry points to a module that are stored in the EXPORT table of a module. They are retrieved during compilation.
(3) Constructors and destructors of global objects: In C++, globally declared objects must be constructed before the main function begins execution. In case of LLVM-clang, for each global variable, the compiler front-end (clang) generates a corresponding `cxx_global_var_init` function. Furthermore, `global_ctors` and `global_dtors`—lists of global constructors and destructors—are generated. These functions are retrieved during compilation.
(4) Call-back functions: In some cases, function pointers are passed from one module to another to be invoked at a later point in execution. For example, the `qsort` function in the standard C library accepts a comparator function that is invoked during comparison of two elements. If unhandled, the invocation of the comparator function results in a U2P transition and leads to stack corruption. Functions that invoke function pointers are identified during compilation, and the potential targets are captured as entry points.

Algorithm 1 Algorithm to partition large frames in a program.

```
1:  procedure PartitionFrames(Program, Alignment)
2:      for each Func in Program do
3:          if Func.LargestVar.Size ≥ Alignment then
4:              return      ▷ Large objects must not be partitioned.
5:          end if
6:          if Func.FrameSize > Alignment then
7:              InsertDummyVars(Func)
8:              SanitizeDummyVars(Func)
9:          end if
10:     end for
11:     return
12: end procedure

13: procedure InsertDummyVars(Func)
14:     used ← CalleeSavedRegs.Size() ▷ Return addr, RBP, other
    callee saved registers.
15:     for each var in Func do
16:         if used + var.Size ≥ Alignment then      ▷ var crosses
    frame boundary, so start a new frame.
17:             NewFrame ← Func.AddAlignedFrame()
18:             dummy = CreateVarAt(NewFrame.start)
19:             Func.dummy_vars.add(dummy)
20:             Func.CurrentFrame ← NewFrame
21:             used ← dummy.Size
22:         end if
23:         Func.CurrentFrame.add(var)
24:         used ← used + var.size
25:     end for
26: end procedure

27: procedure SanitizeDummyVars(Func)
28:     for each dummy in Func.dummy_vars do
29:         NewInsts.add(CreateInst(MOV(dummy,0x0)))
30:     end for
31:     for each Inst in Func do
32:         if Inst.isDirectOrIndirectBranch() then
33:             InsertBefore(Inst, NewInsts)
34:         end if
35:     end for
36: end procedure
```

Entry points are encapsulated in wrapper code that (1) saves the current value of stack pointer, (2) aligns the stack pointer to the next *n*-byte boundary, and (3) calls the corresponding entry point. Upon return from the entry point, the wrapper restores the stack pointer and returns the control back to the unprotected calling module.

5.5 Dynamic allocation using `alloca`

Functions may request for dynamic allocation of stack space through a call to `alloca`. In such cases, the function frame size is unknown at compile time. Stack space is allocated during runtime either as a call to `alloca`, or as an inlined function where an absolute SP-update instruction is used to update the stack pointer. In either

case, after `alloca` completes execution, SPIglass sanitizes the stack pointer by forcing it to the next largest *n*-byte boundary. Deallocation of the stack frame is unaffected because the stack pointer is restored to the caller frame using an absolute SP-update instruction, and like any other deallocation, stack pointer is sanitized by SPIglass. Furthermore, SPIglass can not inject dummy variables to protect functions that use `alloca` if their frame size is larger than *n*-bytes. A detailed discussion is presented in Section 7.2.

5.6 Abnormal Flows

setjmp and longjmp. SPIglass provides implicit support for `setjmp` and `longjmp`. When a call to `setjmp` is encountered, the current state including the instruction pointer and stack pointer are saved into a buffer. Upon execution of `longjmp` the saved values are restored. Because the stack pointer is *n*-byte aligned before the call to `setjmp` (due to **I1**), it remains *n*-byte aligned after `longjmp`.

C++ Exceptions. Both clang and gcc follow Itanium C++ ABI to implement exceptions in C++[1]. After an exception is thrown, `_Unwind_RaiseException` is invoked to perform stack unwinding. It unwinds one frame at a time and updates the stack pointer and program counter until it reaches the frame with the exception handler. This process eventually restores the appropriate execution state including the stack pointer's value before transferring control to the landing pad. As a result, stack pointer integrity is consistent before and after the exception.

JIT code transitions. If a browser is protected with SPI, but the generated Just-In-Time (JIT) code is not, transition from the JIT code to browser code results in a U2P transition. While we do not support JIT code in the current iteration of SPIglass, one solution is to identify and wrap entry points for such U2P transitions. A more effective and robust solution would be to alter the JIT engine to incorporate SPI in the JIT code.

5.7 SPIglass vs PBlocker and EMET

SPIglass accommodates a significantly larger attack space than PBlocker and Microsoft's EMET. First, PBlocker and EMET only defend against inter-stack pivoting where the payload is located outside the stack, whereas SPIglass can defend against all attacks that violate SPI. Second, PBlocker cannot defend against stack-aligned payloads that are injected through stack overflow (Section 4.3 in [32]). SPIglass can protect against such attacks because each potential gadget in modules protected by SPIglass contains code that sanitizes the stack pointer and forces it to *n*-byte alignment, disallowing control transitions between gadgets.

5.8 Defeating COOP

Whenever the number of arguments passed in the main-loop gadget is not equal to the number of arguments accepted by a gadget (virtual function), the stack conservation property is violated, and as such SPIglass will stop such attacks. However, in x86-64, because the first few arguments are passed in registers (rdi, rsi, rdx, rcx, r8, r9), the misalignment of the stack pointer can be avoided for gadgets with 6 or fewer arguments. While SPIglass does not stop such cases where SPI property is not violated, it reduces the total number of gadgets available to COOP and is likely to impact practicability of the attack.

6 EVALUATION

Prototype and Test set. We developed a prototype codenamed SPIglass based on LLVM compiler infrastructure. Though SPIglass works specifically for 32 and 64 bit Intel x86 architectures, it can be ported to other stack-based architectures with minimal changes.

We evaluated SPIglass for runtime and memory overhead for different values of alignment on C and C++ benchmarks from SPEC CPU2006. Although SPI as a property is independent of the programming language, our implementation targets the LLVM compiler's clang front-end, and therefore floating point benchmarks written in FORTRAN are excluded. We also report findings from our experiments on stack utilization, specifically stack frame distribution for Firefox 49.0a1, OpenSSL 1.1.0, Binutils 2.26, and Coreutils 8.25 to provide insights on optimal alignment size. All of our experiments were conducted on systems with quad-core Intel Core i7-4790 @ 3.60GHz, with 16GB RAM, and running Ubuntu 14.04LTS.

6.1 Runtime Overhead

Overhead imposed by SPIglass for different values of alignment for SPEC CPU2006 Integer and Floating point benchmarks is presented in Table 1, and the mean overhead for the benchmarks is depicted in Figure 6. On the one hand, larger alignment increases security because larger objects can be accommodated without a need to partition. But on the other hand, larger alignment could result in a higher performance overhead. An alignment of 2048 results in overhead double that of 512 despite no significant change in code. This is due to the cache misses in the hardware that occur from the large offsets in stack access. In the case of benchmark program `xalancbmk` (an XML processor), the maximum alignment value is *n*=128 bytes. Higher values of alignment require an increase in the default stack space allocated to the process. This is because the `execute()` function in `xalancbmk` recurses through all nodes in the DOM tree, and runs out of stack memory for higher values of alignment. This limitation was overcome by altering an OS-level configuration to increase the default stack space for the process.

Further, we found `dealII`, `sphinx3`, `perlbench`, and `omnetpp` imposed high overhead at 128-byte alignment. The root cause for this counterintuitive behavior was that a significant fraction of frames in these programs were (1) greater than 128 bytes in size (458648, 17126 and 136367 frames for `namd`, `dealII`, and `sphinx3` respectively were 2048 bytes or larger), and (2) belonged to functions that executed frequently. This resulted in high overhead due to partitioning. Our results show that 256-byte alignment provides the best performance. On average, SPIglass introduces a low performance overhead of 5.1% across all benchmarks. Excluding programs that require modifying stack space limit like `xalancbmk`, overhead imposed by SPIglass is only 4.0%.

6.2 Memory Overhead

Memory overhead is presented in Table 3. We implemented a Pintool [21] that monitors the stack utilization by each process in the benchmark programs and reports the maximum stack size used. The

Table 1: Percentage runtime overhead for the SPEC CPU2006 programs when compared to vanilla LLVM 3.7.0 for different alignment values. For xalancbmk, * indicates that the default stack size limit was increased to 512MB to allow for recursion.

	Program	2048	1024	512	256	128
CINT	perlbench	4.6	4.5	0.6	3.7	8.1
	bzip2	1.2	1.0	0.0	0.0	0.1
	mcf	5.2	6.2	1.6	0.0	2.5
	gobmk	1.9	2.3	0.1	0.0	0.0
	hmmer	0.2	0.1	0.1	0.0	0.0
	sjeng	2.2	2.1	2.8	0.1	2.5
	libquantum	1.2	1.6	0.0	0.0	1.1
	h264ref	1.3	1.6	0.0	0.0	0.0
	omnetpp	13.4	10.6	6.7	4.1	6.7
	astar	5.6	4.6	1.1	1.5	3.7
	xalancbmk	28.0*	25.8*	22.0*	21.5*	24.0
FP	milc	2.8	4.0	0.0	3.0	0.8
	namd	0.6	0.5	0.0	0.0	2.6
	dealII	25.0	27.3	24.9	26.6	28.4
	soplex	7.4	6.4	6.0	0.0	0.2
	povray	10.4	8.1	7.4	7.1	6.9
	lbm	2.1	1.6	0.2	0.0	0.3
	sphinx3	2.3	1.7	1.0	0.0	5.5

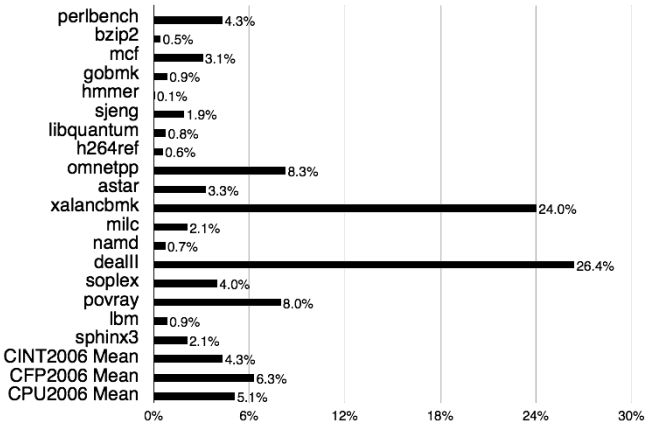

Figure 6: Mean performance overhead for SPEC CPU2006 benchmark for alignments of 128, 256, 512, 1024 and 2048 bytes.

overhead in Table 3 represents the maximum stack size overhead when compared to the stack utilization for programs compiled using mainline LLVM-clang 3.7.0. As expected, mean memory overhead scales linearly with respect to the alignment.

Most functions require frame sizes smaller than 128 bytes (Table 7). In case of `mcf`, most functions require a frame size much smaller than 128 bytes (16 to 32 bytes). Therefore, it shows the highest overhead.

Table 2: SP-update instructions in Metasploit exploits for vulnerabilities reported between 2013 and 2016.

Instruction	CVE
xchg eax, esp	2014-0983, 2014-0307, 2013-3897, 2013-2551, 2013-3918, 2013-3893, 2013-3184, 2013-3163, 2013-3482, 2015-2433
pop esp	2013-3205, 2013-3906
mov esp, src	2013-2492, 2013-0753, 2013-5331, 2012-0439, 2012-594, 2012-0198, 2012-0284, 2013-3205, 2012-0779, 2013-3918, 2014-0322, 2013-1609, 2014-2364, 2014-0783, 2013-3482, 2013-3906, 2013-5447, 2014-0983, 2014-4076
sub esp, offset	2013-1347, 2013-3184, 2013-3163, 2014-2299, 2013-2343, 2013-1892, 2013-5019, 2014-3913, 2013-4800, 2014-0307, 2013-0753, 2013-3897, 2013-5331, 2013-1347, 2013-3205, 2013-2370, 2013-2551, 2013-3918, 2014-0322, 2013-0025, 2013-1017, 2013-3893, 2013-1690, 2013-3163, 2014-2364, 2014-0783, 2014-0782, 2014-3888, 2014-0784, 2013-4988, 2013-3482, 2013-3906

Table 3: SPIglass SPEC CPU2006 memory overhead for different alignments.

	Total Peak Stack Size Overhead (%)				
Benchmark	**2048**	**1024**	**512**	**256**	**128**
perlbench	46.0	24.0	24.0	5.2	2.5
bzip2	209.3	98.9	40.0	14.8	6.1
mcf	2120.4	1020.9	470.9	198.0	60.7
gobmk	454.7	181.2	58.7	10.8	3.9
hmmer	208.0	143.9	42.9	16.1	8.2
sjeng	11.8	5.1	1.8	1.4	0.8
libquantum	669.9	296.4	103.1	45.9	0.0
h264ref	8.1	4.4	1.9	0.8	0.5
omnetpp	9.1	4.3	4.3	0.9	0.2
astar	209.1	105.3	42.4	8.8	3.3
xalancbmk	1968.0	934.2	417.0	159.0	73.0
milc	414.1	163.3	44.7	14.1	12.1
namd	23.9	18.2	16.9	2.6	1.5
dealII	345.9	155.2	70.0	37.0	30.6
soplex	550.2	257.4	126.0	54.2	21.8
povray	322.9	125.3	56.1	33.9	26.5
lbm	221.9	116.3	43.9	17.9	10.2
sphinx3	6.6	2.3	0.6	0.5	0.1
Mean	**590.4**	**279.7**	**122.3**	**46.0**	**17.6**

6.3 Entry Points

We extracted entry points for the SPEC CPU2006 programs, and for each entry point we implemented a wrapper to acclimate the stack pointer. While most programs did not have any entry point other than the `main` function, some C++ programs listed in Table 5 contained global objects that needed non-default construction, destruction or both. These entry points were invoked from libc. Overall, we found the number of entry points to be a very small fraction of the total number of functions in a program.

Table 4: Distribution of frame size for SPEC CPU2006 INT and FP benchmarks. The numbers indicate the number of times a frame of a particular size was created during execution.

	<128	128-256	256-512	512-1024	1024-2048	2048-4096	>4096
perlbench	67,577,603,233	2,430,183,316	3,989,197,903	101,307,234	3,611,416,453	1,915,625	986,731
bzip2	24,909,040,551	6,415	7,407	777,583,860	26,463,756	5,035	116,268
mcf	21,205,433,016	7,695,383	210,515	210,498	49,599	0	0
gobmk	63,379,354,006	1,159,551,112	597,332,823	1,120,457,864	363,415,069	23,021,937	59,197,274
hmmer	3,487,269,303	2,814,778	86,184,643	161,116,193	55,895,832	3,621,190	6,604,171
sjeng	86,167,077,158	567,059,193	302,946,922	232	14,500	0	688,658,543
libquantum	3,442,633,031	102,574	15	880,906,057	6	0	0
h264ref	113,721,593,513	1,770,026,953	510,674	9	26,782	0	0
astar	201,455,384,450	5,512,441	4,349,286	3,361,478	2,413,012	87,822	3
omnetpp	282,543,263,028	1,617,147,891	186,294,667	674,502	1,040,605	5,386	0
xalancbmk	988,032,289,983	1,048,661,060	370,991,236	385,374,116	50,473,966	833,487	1,243
milc	20,477,733,258	254,088,932	5017	89	8,000,802	0	0
namd	24,849,041,564	1,206,788	747,850	739,915	386,210	1	459,648
dealII	1,877,447,583,447	730,875,681	109,176,765	12,904,097	12,400,926	17,051	75
soplex	199,115,726,207	4,725,862	26965328	48,535	9,800,483	0	2
povray	193,195,853,378	7,357,676,846	264,348,601	112,189,636	72,008,245	405	17
lbm	5,313,401	386	255	4	95	0	0
sphinx3	21,493,717,167	627,875,480	1,256,876	1,776,513	32,790,376	2	136,365

Table 5: Non-main entry points for C++ programs in SPEC CPU2006.

Program	Total Functions	# Entry Points	Percentage
xalancbmk	49635	109	0.22
omnetpp	3503	18	0.51
astar	213	2	0.94
h264ref	590	8	1.36
dealII	18725	22	0.12
povray	2013	1	0.05

Table 6: Summary of gadgets and SP-Update Instructions in Coreutils and Binutils.

Suite	Total # gadgets	Absolute SPU	Relative SPU
binutils	170265	15	2406
coreutils	71887	7	2571
libc.so	36515	3	845

6.4 Frame Size Distribution

Function frame size distribution for common programs is presented in Table 7, and frequency of frames of differently-sized brackets for SPEC CPU2006 programs are presented in Table 4. Frequency distribution of the frame size offers vital information in deciding the optimal alignment size. The main consumers of stack space are local variables in functions. As a software development practice, large objects are typically created on the heap, and not on the stack. Therefore, more than 96% of all functions require under 256 bytes of stack frame. As a result, partitioning is not required for most functions when alignment is 256 bytes or higher.

Optimal alignment. Optimal alignment can vary from program to program. However, we found n=256 to offer least performance overhead and covers most functions in a program without the need to partition. Given the low cost of memory, we believe the average memory overhead of 32.6% is reasonable.

6.5 Attack Surface Ananlysis

We examined the number of relative SP-update gadgets SPIglass can eliminate in binutils, coreutils and libc. The results are presented in Table 6. Relative SP-update instructions are far more prevalent than absolute SP-update instructions. Note that SPIglass can defend against attacks that utilize relative SP-update instructions to perform intra-stack pivoting, whereas EMET and PBlocker cannot.

Metasploit exploits: Furthermore, we analyzed the usage of SP-update instructions in the exploits in the Metasploit [2] framework. Results for CVEs reported between 2013 and 2016 are tabulated in Table 2. For vulnerabilities with CVEs between 2010 and 2016, we found 84 exploits that used `sub esp, offset` relative SP-update instruction whereas 60 exploits utilized absolute SP-update instructions `xchg eax, esp` and `pop esp` to carry out the exploit. SPIglass can successfully defend against attacks that utilize relative SP-update instructions in pivoting.

7 DISCUSSION

7.1 Effect on Recursion

While SPIglass does limit the total possible depth in non-tail recursion, it has no impact on tail recursion. During tail recursion, the compiler does not create a separate function frame for each instance of recursion. Instead, it reuses the same stack frame for all recursive invocations, introducing unconditional `jmp` instructions to mimic a recursive call. Because the same stack frame is reused, the frame size overhead introduced by SPIglass does not accumulate over all the instances of recursion.

Table 7: Percentage distribution of frame size for 49.0a1, coreutils 8.25, binutils 2.26 and OpenSSL 1.1.0.

Program	size < 128	128 ≤ size < 256	256 ≤ size < 512	512 ≤ size < 1024	1024 ≤ size < 2048	size ≥ 2048
Firefox	92.33	4.52	1.98	0.80	0.21	0.16
Coreutils	80.25	10.34	5.96	1.47	0.5	1.48
Binutils	92.3	5.58	1.47	0.31	0.16	0.19
OpenSSL	92.74	4.38	1.21	0.47	0.57	0.63

Table 8: Percentage distribution of variable size for Firefox 49.0a1, coreutils 8.25, binutils 2.26 and OpenSSL 1.1.0.

Program	size < 128	128 ≤ size < 256	256 ≤ size < 512	512 ≤ size < 1024	1024 ≤ size < 2048	size ≥ 2048
Firefox	96.64	2.14	0.84	0.14	0.10	0.14
Coreutils	92.14	5.15	0.74	0.50	0.23	1.24
Binutils	99.48	0.31	0.05	0.0	0.10	0.09
OpenSSL	97.34	0.76	0.61	0.16	0.53	0.61

For non tail-call recursion, total possible depth of recursion reduces with the alignment used. However, the reduction can be compensated by increasing the size of stack segment proportional to the recursion depth.

7.2 Large Contiguous Allocations on Stack

Though uncommon, it is possible for a large object (e.g., array or structure) whose size is greater than the alignment to be allocated on the stack either statically (known at compile time), or dynamically (using `alloca`). The variable size distribution for Firefox, Coreutils, Binutils and Openssl is presented in Table 8. Most stack variables are under 256 bytes in size. Partitioning large objects (e.g., structures > 2048 bytes in size) could lead to data corruption. When sizes of such objects exceed the alignment size, there is a possibility that an attacker injects payload into such large objects, and performs a pivot operation to execute the payload. A potential solution is to relocate large objects to the heap, and rely on **P1** to stop pivoting. A similar technique is implemented by StackArmor [9].

7.3 Inlined Assembly

The ability to lay out the stack frame is key to enforcing SPI, particularly invariant **I2**. Modern compilers (GCC and LLVM) fix the stack frame during the execution of a function. Any save/restore operations are performed in the prologue and epilogue of a function. However, it is possible for hand-crafted assembly code (possibly inlined) to use `push` and `pop` instructions within function body. Such inlined assembly code can hinder frame size calculations. Our investigation did not reveal any legal compilation scenarios (on gcc and LLVM) that incorporated implicit SP-update instructions within function body.

8 RELATED WORK

CFI-based defenses. Numerous efforts [3, 24–26, 33, 39] have attempted to defeat code-reuse attacks by enforcing various forms of CFI. These solutions extract the CFG and insert inlined reference monitors either by relying on source code and debugging information, or by analyzing the binary itself [44, 45]. Variations of CFI targeting either performance [7, 33, 43], or security [22, 40] have been proposed. Fundamentally, completing the CFG is a hard problem, and improving the precision of indirect branch resolution is an ongoing direction of research [41]. SPIglass is orthogonal to the above approaches, and in combination with CFI, SPI provides stronger overall security.

ASLR-based defenses. ASLR [4, 38] was introduced as a means of preventing attackers from reusing exploit code effectively against multiple instantiations of a single vulnerable program. Recent efforts such as binary stirring [42] and TASR [5] focus on increasing the effectiveness of ASLR by increasing re-randomization frequency. Redactor [10, 11] uses a combination of compiler transformations and hardware-based enforcement to mark pages as execute-only, thereby defeating the objective of memory disclosures. Techniques that combine CFI and ASLR have also been proposed [23]. While ASLR can strengthen SPI by making it hard to excavate gadgets, SPI is not dependent on ASLR.

Stack-based defenses. Microsoft's EMET [14] defends against stack pivoting by checking the stack pointer within sensitive APIs (like VirtualProtect) to ensure that it lies within the stack region of a thread. However, DeMott demonstrated that the difference between time-of-check to time-of-use of the stack pointer can be exploited for practical attacks against EMET [13]. PBlocker [32] takes EMET's idea one step further and inserts assertion checks to ensure that the stack pointer lies within the stack region immediately after each SP-update instruction.

Fu et al. [16] introduced Slick, a solution that makes use of exception handling metadata available in a binary to detect stack layout corruptions caused by an ROP attack. Similar to MS-EMET, there is a window of time between stack corruption and Slick's checks that can give the attacker an opportunity to hide her or his trail. Slowinski et al. [37] introduces a memory corruption defense that addresses diversion of control-flow through illegal data modification detection. Their solutions assign an id (color) to a memory object and a pointer. A pointer is only allowed access to a memory object with matching id. SPI as a property is associated with the stack pointer and not the contents of the stack. SPIglass strictly monitors and enforces stack pointer invariants immediately after each explicit stack pointer update.

Other defenses. Kuznetsov et al. [19] proposed Code-Pointer Integrity (CPI), a policy guaranteeing the sanity of code pointers such as function pointers and saved return addresses throughout program execution. CPI partitions memory into safe and regular

regions. By limiting the subset of memory objects that are protected, CPI limits the amount of instrumentation compared to CFI schemes and thus achieves a performance benefit. SPI is orthogonal to CPI, and will harden CPI. Dynaguard by Petsios et al. [30] resists brute-force canary attacks like BlindROP [6] through a per-thread run-time update of canary values in all stack frames. SPI is fundamentally different yet complementary to these defenses.

9 CONCLUSION

We introduce Stack-Pointer Integrity, a program integrity property that aims to complement CFI and other modern defenses. We present SPIglass, an LLVM-based implementation of SPI that makes a memory trade off in order to improve security.

It provides interoperability, and strong security orthogonal to CFI. We find the stack alignment size of 256 to be optimal. We opensource SPIglass.

10 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their feedback. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929 and National Science Foundation Award #1566532. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2015. Exception Handling in LLVM. http://llvm.org/docs/ExceptionHandling.html. (2015).
[2] 2017. Metasploit penetration testing framework. http://http://www.metasploit.com/. (2017).
[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 340–353.
[4] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security*, Vol. 3. 105–120.
[5] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 268–279.
[6] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking Blind. In *IEEE Symposium on Security and Privacy (SP'2014)*. IEEE, 227–242.
[7] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating Code-reuse Attacks with Control-flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. 353–362.
[8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 161–176.
[9] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. 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)*.
[10] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*. 763–780. https://doi.org/10.1109/SP.2015.52
[11] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 243–255.
[12] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, Vol. 15.
[13] Jared DeMott. 2014. Bypassing EMET 4.1. https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf. (2014).
[14] J. DeMott. 2015. Bypassing EMET 4.1. *IEEE Security Privacy* 13, 4 (July 2015), 66–72. https://doi.org/10.1109/MSP.2015.75
[15] Erica Eng and Dan Caselden. 2015. 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. (2015).
[16] Yangchun Fu, Jungwhan Rhee, Zhiqiang Lin, Zhichun Li, Hui Zhang, and Guofei Jiang. 2016. Detecting Stack Layout Corruptions with Robust Stack Unwinding. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'16)*. Paris, France.
[17] Robert Gawlik and Thorsten Holz. 2014. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of 30th Annual Computer Security Applications Conference (ACSAC'14)*.
[18] Enes Göktaş, Elias Anthanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)*.
[19] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[20] Julian Lettner, Benjamin Kollenda, Andrei Homescu, Per Larsen, Felix Schuster, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Michael Franz. 2016. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 209–221. https://www.usenix.org/conference/atc16/technical-sessions/presentation/lettner
[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 190–200.
[22] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 941–951. https://doi.org/10.1145/2810103.2813676
[23] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. 2015. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*.
[24] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.
[25] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of 21st ACM Conference on Computer and Communication Security (CCS '14)*.
[26] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 914–926.
[27] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking holes in information hiding. In *USENIX Security*.
[28] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 49–58.
[29] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming using in-place Code Randomization. In *IEEE Symposium on Security and Privacy (SP'2012)*. 601–615.
[30] Theofilos Petsios, Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2015. DynaGuard: Armoring Canary-based Protections against Brute-force Attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 351–360.
[31] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. 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)*.
[32] Aravind Prakash and Heng Yin. 2015. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 111–120.
[33] Rui Qiao, Mingwei Zhang, and R Sekar. 2015. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 101–110.
[34] Ahmad-Reza Sadeghi, Lucas Davi, and Per Larsen. 2015. Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future?. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 55–61.
[35] Hovav Shacham. 2007. 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*. ACM, 552–561.
[36] Felix Shuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-reza Sadeghi, and Thorsten Holz. 2015. 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).

[37] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2012. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 125–137. https://www.usenix.org/conference/atc12/technical-sessions/presentation/slowinska

[38] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003).

[39] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)*. 941–955.

[40] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 927–940.

[41] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of 37th IEEE Symposium on Security and Privacy (Oakland'16)*.

[42] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. 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)*. ACM, 157–168.

[43] Chao Zhang, Scott A Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. 2016. VTrust: Regaining Trust on Virtual Calls. In *Symposium on Network and Distributed System Security (NDSS'16)*. https://doi.org/10.14722/ndss.2016.23164

[44] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'13)*. 559–573.

[45] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)*. 337–352.