May 2015

Binary Program Integrity Models for Defeating Code-Reuse Attacks

Aravind Prakash
Syracuse University

ABSTRACT

During a cyber-attack, an adversary executes offensive maneuvers to target computer systems. Particularly, an attacker often exploits a vulnerability within a program, hijacks control-flow, and executes malicious code. Data Execution Prevention (DEP), a hardware-enforced security feature, prevents an attacker from directly executing the injected malicious code. Therefore, attackers have resorted to code-reuse attacks, wherein carefully chosen fragments of code within existing code sections of a program are sequentially executed to accomplish malicious logic. Code-reuse attacks are ubiquitous and account for majority of the attacks in the wild. On one hand, due to the wide use of closed-source software, binary-level solutions are essential. On the other hand, without access to source-code and debug-information, defending raw binaries is hard.

A majority of defenses against code-reuse attacks enforce "control-flow integrity", a program property that requires the runtime execution of a program to adhere to a statically determined control-flow graph (CFG) – a graph that captures the intended flow of control within the program. While defenses against code-reuse attacks have focused on reducing the attack space, due to the lack of high-level semantics in the binary, they lack in precision, which in turn results in smaller yet significant attack space.

This dissertation presents program integrity models aimed at narrowing the attack space available to execute code-reuse attacks. First, we take a semantic-recovery approach to restrict the targets of indirect branches in a binary. Then, we further improve the

precision by recovering C++-level semantics, and enforce a strict integrity model that improves precision for virtual function calls in the binary. Finally, in order to further reduce the attack space, we take a different perspective on defense against code-reuse attacks, and introduce Stack-Pointer Integrity – a novel integrity model targeted at ensuring the integrity of stack pointer as opposed to the instruction pointer.

Our results show that the semantic-recovery-based approaches can help in significantly reducing the attack space by improving the precision of the underlying CFG. Function-level semantic recovery can eliminate 99.47% of inaccurate targets, whereas recovering virtual callsites and VTables at a C++ level can eliminate 99.99% of inaccurate targets.

BINARY PROGRAM INTEGRITY MODELS FOR DEFEATING CODE-REUSE ATTACKS

by

Aravind Prakash

B.E. Visvesvaraya Technological University, Belgaum, India, 2004
M.S. University of Miami, FL, 2009

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical & Computer Engineering

Syracuse University
May 2015

To my parents and my wife Pallavi...

ACKNOWLEDGMENTS

I would like to express my deep and sincere thanks to people who made this dissertation possible.

I would like to thank my advisor, Dr. Heng Yin: Through your hard work, perseverance and emphasis on "good research practices", you have not only played an instrumental role in carving the researcher out of me, but also have taught me how to be a good advisor. Specifically, you taught me how to distinguish between engineering and research elements of a project, and to succinctly capture the research contributions. For this, I am eternally grateful.

I would also like to thank my ex-labmates Lok and Eknath, and current labmates Mu, Qian, Xunchao, Andrew, Yue, Rundong, Jinghan, Pallavi and Abhishek, who have been tremendously helpful. While some of them have directly contributed to my research, all of them have helped me address concerns in early stages of projects by providing timely feedback and reviews of my work.

Most importantly, I would like to thank my family. To my parents: You taught me that education is a basic necessity in life, and put my needs ahead of your own. This is a fruit of your sacrifices. To my wife, Pallavi: You, as a graduate student yourself, were able to relate on a daily basis, to the stress and pressure that accompanies a PhD. You were always there for me. Without you, this would not be possible.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Several of the critical public and private infrastructures – that define life as we know it – are comprised of interconnected software components. In fact, we are so dependent on software that their safety and reliability directly impact human life. Over the last few years, attacks against software have increased at an alarming rate. With wide deployment of Data Execution Prevention (DEP) [1] in the hardware, attackers have resorted to reusing code fragments in existing code sections of a binary. Such attacks are called code-reuse attacks. There has been a systematic transition from individuals indulging in mischief for the purpose of attention-gaining, to well organized sometimes state sponsored syndicates capable of executing complex attacks (e.g., Sony Hack [2], Stuxnet [3]).

Now more than ever, there is a need to automatically protect binaries from attacks. Particularly, with wide use of closed source software, binary-only solutions are a necessity. While source code based approaches can leverage the rich program semantics available from source code (e.g., [4], [5], [6]) binary-only defenses are hard, and introduce key challenges.

State-of-the-art binary-level solutions enforce program integrity policies in order to provide principled defense. This dissertation focuses on providing precise program integrity models on the binary with focus on reducing the attack space for code-reuse attacks.

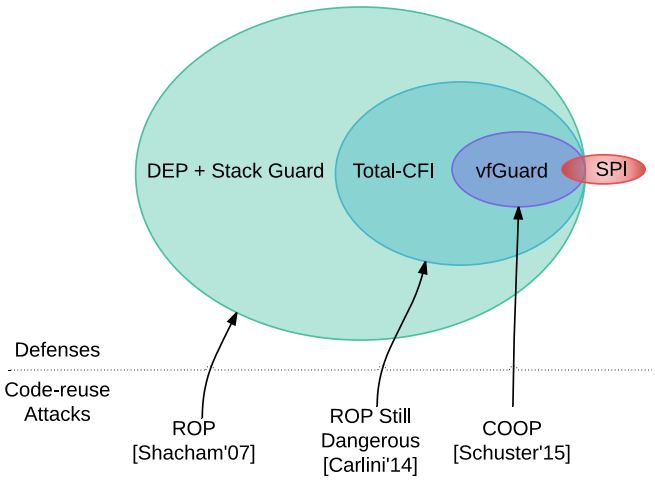

Fig. 1.1.: Attack space reduction

1.1 Dissertation Overview

A program integrity solution comprises of two components: Policy Generation and Policy Enforcement. While Policy Generation aims at generating more precise policies that can thwart code-reuse attacks, enforcement focuses on enforcing the derived policy in a performance-friendly manner.

Our contribution in policy generation is summarized in Figure 1.1. The figure is annotated with code-reuse attacks that are capable of targeting each level of defense. We present `Total-CFI` and `vfGuard`, two solutions that reduce the attack space of code-resue attacks by enforcing precise CFI models. However, over time, attacks have evolved and are now capable of defeating most state-of-the-art solutions. In our final solution presented in this dissertation, we present Stack-Pointer Integrity as a novel program integrity model. We approach the problem from the perspective of stack-pointer, which assumes the role of a program counter (as opposed to instruction pointer) during a code-reuse attack. While

SPI further reduces the attack space, it provides even stronger security in combination with CFI. We show that recovering function level semantics and C++ level semantics can improve the precision by 99.473% and 99.999% respectively.

In order to enforce the generated policy, we discuss three enforcement models. First, `Total-CFI` uses *whole system monitoring* wherein all the processes in the system are simultaneously monitored for violations. Second, `vfGuard` uses *process level emulation*, where policies are enforced by emulating each process separately. Finally, in SPI, we use *binary instrumentation* to modify a binary to embed the security primitives within the binary. We show that SPI can defend against inter-stack pivoting with a small overhead of 1%.

1.2 Our Thesis

Recovering high-level semantics from the binary can aid in reducing the attack space in code-reuse attacks. Enforcing the integrity of stack pointer further reduces the attack space.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides the high-level background essential in understanding this dissertation. Chapter 3 presents three policy generation solutions `Total-CFI`, `vfGuard` and SPI that successively decrease attack space. Chapter 4 presents three enforcement models. Chapter 5 discusses the limitations of this thesis and future work, and finally Chapter 6 concludes.

2. BACKGROUND

This section gives a survey of state-of-the-art in attacks and defenses on the binary, and program integrity models. The section ends with shortcomings of the practical implementations of the integrity models, which forms the basis for this dissertation.

2.1 Binary-Level Attacks

Binary-level attacks alter the target program's control data (e.g., function pointer, return address) directly in the memory in order to achieve arbitrary code execution. With the advent of hardware mechanisms that prevent data execution (e.g., DEP, NX), attackers can no longer execute the injected code after exploiting the vulnerability. Therefore, attacks that reuse existing code – known as code-reuse attacks – are on a rise. Particularly, Return-Oriented Programming (ROP) [7] gleans short 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 [7] showed that ROP is turing complete. Since, several real world attacks employ ROP to bypass DEP. ROP as an attack mechanism is so popular and mature that there exist tools to automatically extract gadgets [8] from programs and compile them [9] (i.e., chain them) to implement program logic.

More recently, Carlini and Wagner [10], and Göktaş et al. [11], demonstrate practical attacks that reuse code fragments that span several instructions, sometimes entire functions. Such attacks not only bring the traditional definition of a gadget into question, but also highlight that defenses lack precision.

2.2 Binary-Level Defenses against Code-Reuse Attacks

2.2.1 Artificial Diversification

The goal of artificial diversity is to randomize and hide the location of a program's code, data, stack, heap, etc [12–16]. STIR [17] performs static instrumentation to generate binaries that self-randomize every time the binary is loaded. Isomeron [18], 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 [19] [15] provide intra-modular ASLR using code-islands in order to defend against chained return-to-libc attacks, wherein they identify and randomize into isolated code blocks. That is, they not only randomize the base addresses of memory mappings, but also randomize the relative offsets between each functions.

Artificial-diversity-based defenses are fundamentally susceptible to disclosure attacks, and are not always an effective defense [20]. Moreover, they are also susceptible in cases where an attacker introduces the code (with predictable layout) that s/he can reuse [21].

2.2.2 Gadget Elimination

Two main works: in-place code randomization [22]and G-Free [23] have been proposed to eliminate gadgets. G-Free is a compiler extension that compiles the source-code to generate gadget-free binaries. Pappas et al. [22] operate directly on the binary by first identifying the gadgets, and then eliminating them by performing in-place semantic-preserving modifications. Fundamentally, these solutions suffer from two limitations:

1. **Weak Definition of Gadget:** Per these solutions, a *short* sequence of instructions terminated by a `ret` instruction is treated as a gadget. However, in principle, a sequence of instructions of *any* length can be used as a gadget as long as the attacker can negate the undesirable side effects of using a gadget by using appropriate compensatory gadgets.

2. **Lack of Coverage:** Due to their intrusive nature, and the large number of gadgets to eliminate [7], they can only eliminate a fraction of all the gadgets in the binary. For example, Pappas et al. [22] eliminate 76.9% gadgets in Windows XP and 7, leaving an attacker with 23.1% or 6.3M gadgets to take advantage of.

2.2.3 Control-Flow Integrity

Control-flow integrity (CFI) was first proposed by Abadi et al. in 2005 [24] as part of the compiler framework to automatically place in-line reference monitors in the emitted binary code to ensure the legitimacy of control transfers. CFI as a program property

dictates that software execution must follow a path of a Control-Flow Graph [25] determined ahead of time. Since then, a great deal of research efforts have built on top of it. Some efforts extended the compiler framework to provide better CFI protection.

Compile-Time Defenses MCFI [26] enables a concept of modular control flow integrity by supporting separating compilation. KCoFI [27] provides control flow integrity for commodity operating system kernels. RockJIT [28] aims to provide control flow integrity for JIT compilers.

Other efforts are made to enforce control-flow integrity directly on binary code. Efforts such as PittSFIeld [29] and CCFIR [30] enforce coarse-grained policy by aligning code. Based on a CPU emulator, MoCFI [31] rewrites ARM binary code to retrofit CFI protection on smart phone. While Opaque CFI [32] combines coarse-grained CFI and artificial diversification in order to render disclosure attacks harder, it introduces other problems like large space overhead, lack of fine-grained diversification, etc.

Several solutions have also been proposed at a source-code level [4, 6]. Lhee and Chapin [33] associate type information with buffers during compile time in order to prevent buffer overflows. Code-Pointer Integrity (CPI) [34] protect pointers to code, etc

Run-Time Defenses Runtime defenses monitor the execution during run time, and identify anomalies in the control flow. Xu and Chapin [35] generate and validate a policy that captures the legitimate control flows leading up to each system call. More recently, kBouncer [36] and ROPecker [37] periodically examine the control flow to detect anomalies. Particularly, they associate heuristics to "gadgets", and by examining the "Last Branch

Record" (LBR) – a hardware feature provided by the Intel CPU, they identify potential attacks. All of these defenses have been defeated by the attacks proposed in [10] and [38]. Further, PointerScope [39] performs exploit diagnosis by automatically inferring and detecting type violations during execution. As a runtime diagnosis system, it imposes severe performance overhead.

2.3 Shortcomings of Current Binary-Level Defenses

Shortcomings of state-of-the-art binary-level defenses against code-reuse attacks are tabulated in Table 2.1.

Table 2.1: Shortcomings of current binary-level defenses against code-reuse attacks

Category (Representative Solution)	**Parameter**	**Status**	**Reason**
CFI (BinCFI [40])	Precision	Low	Approximate CFG
	Deploy-ability	Not incrementally deployable, Requires a priori knowledge of vulnerable process	Incomplete CFG
	Runtime Performance overhead	*> 8%*	Approximate CFG
	Gadget Definition	Weak	Does not support large, function-entry, or call-preceded gadgets
Artificial Diversity ASLR [41]	Resilience	Partial	Vulnerable to memory disclosure attacks and Just-in-time injected gadgets
	Diversification	Partial	In practice, not all modules are diversified.
Gadget Elimination (G-Free [23])	Coverage	Low	Too many gadgets to eliminate

2.3.1 Precision

Binary-level CFI-based defenses (e.g., BinCFI [40], CCFIR [30]) suffer from low precision due to their coarse-grained nature. Without source code, these solutions take a conservative approach and include redundant edges in the CFG, which leads to an exploitable attack space. Practical attacks against such defenses were demonstrated by Carline and Wagner [10], and Göktaş et al. [11].

2.3.2 Deployment

Module-Level Deployment Key to CFI-based defenses is that they rely on recovering the CFG of the program. However, when a module is compiled, it may not be possible to ascertain *all* the other modules that would be using the module. This leads to the *incomplete CFG* problem. Due to this problem, modules can not be selectively protected unless all the modules that are used by a program are protected.

Some solutions [26] have been proposed to extend the CFG, and hence the CFI protection at runtime, however, these solutions require changes to the loader, which may not always be possible.

Sytem-Wide Deployment In order to defend against attacks in a performance-efficient manner, runtime CFI-based defenses must know the precise processes to monitor. However, vulnerabilities could exist in multiple modules spanning multiple processes. This leads to a scalability problem wherein, CFI-based runtime defenses can not scale with the running

processes in the system. This is particularly important to defend against complex malware like Stuxnet [3] that exploit multiple vulnerabilities in across the system.

3. ATTACK SPACE REDUCTION

In this chapter, we describe each of the three solutions that propose stringent models in order to reduce the attack space.

3.1 Precise CFI Model through Recovery of Function Semantics

There are several challenges to enforce CFI in practice. The key challenge is that CFI requires a *complete* control-flow graph (CFG), which is hard to compute without source code. In this section, we present `Total-CFI`, a tool that recovers function-level semantics from a binary in order to improve the precision of CFI policy [42].

3.1.1 `Total-CFI` Overview

`Total-CFI` leverages full system emulation to monitor the guest operating system from its inception. Schematic overview of `Total-CFI` can be found in Figure 3.1. At a high level, `Total-CFI` consists of 2 components, *Policy Generation* and *Policy Enforcement*. In this chapter, we confine the scope to policy generation, we discuss policy enforcement in Chapter 4.

Model Generation The model generated by `Total-CFI` consists of a static and a dynamic component. The static policy comprises of a whitelist that succinctly captures all

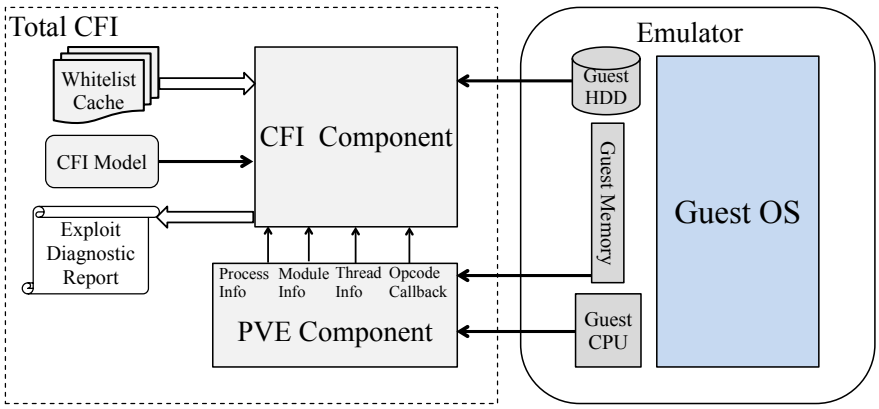

Fig. 3.1.: Architecture Overview of `Total-CFI`

the allowable targets for indirect `call/jmp` instructions. A whitelist for a specific module is made up of all the statically determinable target addresses for indirect control flow in the module, such as the elements of relocation table and the export table. A whitelist for a process is a union of all the whitelists of all the modules loaded in the process address space. Whereas the dynamic or the runtime component of the policy intercepts each `call` and `ret` instructions. The policy requires that each `ret` instruction return to an instruction succeeding the previously executed `call` instruction.

3.1.2 Static Policy

Exploits alter the normal control flow by manipulating the derived code addresses (e.g., function pointers). `Total-CFI`'s CFI model is based on the observation that most of the control flow is restricted by a pre-determined subset of code that forms the entry point of branch targets. For example, targets of `call, jmp` instructions must adhere to the statically determined call graph, and can not branch to *any* arbitrary location. Based on

this observation, a program is statically analyzed to generate a whitelist, a list of allowable targets for each indirect `call/jmp` instruction.

Target Whitelist The addresses within the relocation table and the export table of the binary constitute the module whitelist. With compatibility in mind, most modern binaries are compiled to be relocatable [43]. When the loader cannot load a binary at its default location, it performs relocation. The loader refers to the relocation table and fixes the addresses of the entries in the relocation table. Indirectly addressable code must be relocatable. Similarly, export table contains the functions that a given module exposes for use by other modules. Addresses of such functions are resolved at runtime based on the actual load address of the dependent modules. Therefore entries of the relocation table and the export table of a module together form valid indirect branch targets for a module.

Irrespective of the guest OS being executed, the loader first needs to load the entire module binary to memory and perform relocation fix-ups (if any) before transferring control to the module. However, when the control reaches the module entry, it is possible that the guest OS memory manager has flushed the page containing the relocation table from the memory. To optimize the whitelist extraction, `Total-CFI` first tries to retrieve the relocation and export tables corresponding to a module directly from the guest memory. If the pages corresponding to relocation and export table are paged out, `Total-CFI` accesses the binary file corresponding to the module on the guest file system and extracts the relocation table and export table from the binary.

For each process in the system, `Total-CFI` maintains a sorted array of loaded modules in the process address space. It also maintains a hashtable that maps the base address of a

module to the whitelist corresponding to the module. When a module is loaded, `Total-CFI` first checks the whitelist cache for the whitelist corresponding to the module, only if the whitelist is not present, it constructs the whitelist for the module and adds the whitelist to the whitelist cache.

When `Total-CFI` encounters an indirect `call` or `jmp` instruction, it performs a binary search for the target address in the loaded modules array of the process. Here, the binary search returns a negative insertion point if the search failed. If the returned search value is an even negative index, then the target address does not belong to any of the modules and is treated as a violation of CFI model. However, if the return value is an odd negative index, the target address belongs to the module with base address equal to the address at index - 1 in the loaded module array. Note that it is not possible for the return value to be positive since the target address cannot be equal to the start address or the end address of a module. Therefore, if the return value is non-negative, the address is considered not to be present.

The whitelist lookup is performed with a time complexity of $\lg(n)$, where n is the number of modules in the process address space. Although maintaining a single hash-table for each process with all the whitelists corresponding to all the modules in its address space will suffice, such an approach leads to severe memory overhead due to redundancies, because several common modules (like NTDLL.DLL, KERNEL32.DLL, etc.) will be present in every process whitelist.

Whitelist caching The whitelist for a binary is statically determined and therefore remains the same across different execution instances. As an optimization, the whitelist is

generated only once per binary file and the generated whitelist is stored in the whitelist cache as a [*file's md5 checksum, whitelist*] pair. When a new file is loaded, as an optimization, `Total-CFI` first checks the whitelist cache to determine if the whitelist has already been extracted, only if the file is being encountered for the first time, `Total-CFI` extracts the whitelist and adds the whitelist to the whitelist cache.

3.1.3 Dynamic Policy

Runtime model for CFI enforcement in `Total-CFI` is based on the fact that a `ret` instruction must return to an address succeeding a `call` instruction that was previously encountered.

`Total-CFI` maintains two shadow call stacks per executing thread in the system. One stack shadows the user level stack of the thread and the other shadows the kernel level stack. Whenever a `call` instruction is encountered, `Total-CFI` pushes the return address to the corresponding shadow stack (kernel level shadow stack if operating in kernel mode and user level shadow stack if operating in user mode) of the currently executing thread.

When a `ret` instruction is encountered, `Total-CFI` pops the target address of the return instruction from the appropriate stack of the currently executing thread. If the target address is not found on the shadow stack and the target address does not belong to dynamically generated code, `Total-CFI` infers the `ret` instruction to be a part of a potential exploit.

Shadow Call Stack To keep track of the `call-ret` pairs during the execution of a thread in the guest OS, `Total-CFI` maintains two shadow call stacks per thread - one for

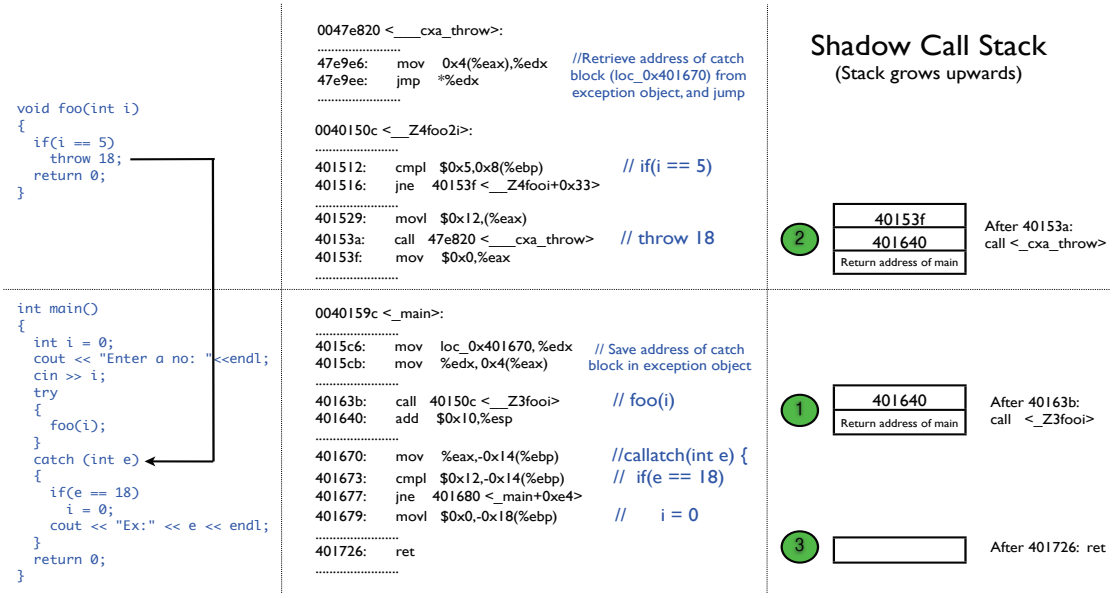

Fig. 3.2.: Shadow Call Stack Behavior During a C++ Exception

user mode execution and one for kernel mode execution. When a `call` or `ret` instruction is executed, the context is identified. The context constitutes the process, thread and the user/kernel mode the instruction was executed in. From the context, `Total-CFI` identifies the appropriate shadow call stack. Then, if the instruction is a `call` instruction, it pushes the address of the succeeding instruction on to the identified shadow stack. Conversely, if the instruction is a `ret` instruction, `Total-CFI` pops the target address off the shadow stack. If the address is not present in the shadow stack, `Total-CFI` reports an exploit.

Though strict pairing between `call-ret` pairs account for a majority of control transfers, there are certain special control transfers that make CFI enforcement via shadow call stack monitoring challenging. Below, we consider such special control flow scenarios.

Handling of Exceptions: Exception handling is a mechanism to handle anomalous events that often change the normal control flow of a program. Figure 3.2 describes the handling of such exceptions by `Total-CFI`. Column 1 lists the source code of a program

that raises and handles an exception. Column 2 lists the simplified version of the corresponding code in assembly, obtained when the code is compiled using the MinGW-g++ cross compiler. The exception handler or the `catch` block is relocatable and hence appears as an entry in the relocation table. During compile time, the compiler stores the address of such a block in the exception object. At runtime, when an exception is thrown, the throw statement translates to a call to *__cxa_throw*, which in turn retrieves the address of the `catch` block from the exception object, rewinds the stack and transfers control to the `catch` block via an indirect jump. Column 3 of Figure 3.2 shows the contents of shadow stack at different stages during the program execution. On one hand, during the `jmp` instruction, `Total-CFI` verifies that the branch address is a part of the program's whitelist and lets the instruction pass, but on the other hand, when the *main* function returns (stage 4 in Column 3 of Figure 3.2), the `Total-CFI` recognizes that the return address is not at the top of the shadow stack and therefore pops all the items up to and including the return address of *main* function from the top of the stack.

Handling of `setjmp/longjmp`: In C and its flavors, `setjmp` and `longjmp` are used to save and restore the CPU environment respectively, in order to transfer control to a predetermined location. During `setjmp`, the environment including the contents of the CPU registers are cached in a user provided buffer, and during `longjmp`, the CPU register contents are restored from the buffer. Upon encountering a `setjmp`, `Total-CFI` records the value of the program counter where the control will be transferred to during `longjmp`. When a `longjmp` is encountered, `Total-CFI` verifies the target address to be the same as the value of the program counter as recorded during the previous `setjmp` in the current execution context. Mismatch in the target address is flagged as a potential exploit.

Kernel mode to User mode call backs: Typically, the control transfers from user mode to kernel mode happen through the `sysenter` and `int` instructions, and back from kernel mode to user mode via `sysexit` and `iret` instructions respectively. However, in Windows, NTDLL maintains a set of entry points that are used by the kernel to invoke certain functionality on behalf of the user mode [44]. Some such NTDLL APIs are: *KiUserExceptionDispatcher, KiUserApcDispatcher, KiRaiseUserExceptionDispatcher* and *KiUserCallbackDispatcher*. They are used by the kernel as a trampoline to invoke functionality in the user mode. Kernel saves the processor state and alters the thread stack to accomplish such a call. When the kernel alters the execution of a thread and transitions to user mode, the return address may not coincide with the expected return address at the top of the stack. To address this problem, for every stack in the system, `Total-CFI` maintains a hash-table wherein, for every `ESP` register value encountered during a call instruction as key, it stores the position of the entry in the stack as value. When a return instruction is encountered, it first checks the `ESP` register's value in the hash-table to find the position on the shadow stack and then, pops all the elements up to that position off the stack. Such an approach is reasonable because, the stack is dictated primarily by the `ESP` register and a rewind of the ESP register would imply a clean-up of the stack. If the `ESP` register value is not found in the hash-table, the instruction is flagged as a potential exploit.

3.1.4 Dynamically Generated Code

The key challenge with dynamically generated code (e.g., JIT code), is that the code can not be statically analyzed and therefore, the whitelists can not be generated.

Execution of dynamically generated code portray the following characteristics:

(i) Firstly, the page containing the dynamically generated code must be written to memory and made executable (particularly on DEP-enabled systems) before it is executed.

(ii) Secondly, control transitions from non-dynamic to dynamic code follow a finite pre-set path.

At runtime, `Total-CFI` tracks the entries in the code and the write caches of Translation Lookup Buffer (TLB) of the CPU to identify dynamically generated code. If an entry in the write cache of the TLB appears in the code cache of the TLB, the entry is identified as dynamically generated code.

Initially, `Total-CFI` is trained to accumulate the possible control paths that lead to dynamically generated code in a particular application. This is done by recording the shadow stacks for the valid control flows that lead to dynamically generated code. An intersection of such paths is used as a signature that is enforced during execution. Here, it is possible that the dynamic code generation library is loaded at different locations on each instance it is loaded. Therefore, `Total-CFI` maintains the signature as a [*module:offset*] pair to validate across load instances.

During normal execution, when `Total-CFI` encounters a branch target that is not in the whitelist, it first checks if the target belongs to dynamically generated code, next it checks the shadow call stack to check if the shadow call stack satisfies the dynamic code signature for the application.

3.1.5 Non-relocatable binaries

Though most binaries are relocatable, some legacy code can be non-relocatable. In such cases, `Total-CFI` statically analyzes the binaries to extract all the statically identifiable addresses - the ones that either occur as constant address operands in the disassembly or the ones that have a function prologue. Though this approach includes addresses which may not be valid targets, such a conservative approach will reduce false positives.

3.1.6 Branch Tables or Jump Tables

A jump table is an array of function pointers or an array of machine code jump instructions. Calls to the functions (or code blocks) in the array are made through indirect addressing using the base address of the jump table and the offset of the desired code block in the table. We make two key observations about jump tables:

1. The base address of a jump table must be relocatable and therefore contains an entry in the relocation table.

2. Every entry in the jump table must point to a valid code block.

`Total-CFI` takes a liberal approach to handle jump tables. For every entry in the relocation table, `Total-CFI` checks if the content at that address points to code, if so, it treats it as a potential base address of a jump table. It traverses the table for consecutive entries that point to code and adds them to the whitelist.

```
00000a44 <foo>:
.....................
 a69:  call   9a7<__i686.get_pc_thunk>
 a6e:  add    $0x1586,%ebx  //Offset of f()
 a74:  call   *%ebx
.....................
000009a7 <__i686.get_pc_thunk>:
 9a7:  mov    (%esp),%ebx
 9aa:  ret
```

Fig. 3.3.: Position Independent Code

3.1.7 Position Independent Code (PIC)

The addressing in PIC does not rely on any particular position in the program address space. Conceptually, PIC identifies the current value of Program Counter (PC) and addresses different code blocks as offsets from the PC. Figure 3.3 shows a typical example of PIC. The current version of `Total-CFI` does not support PIC, however it is possible to parse the binary to scan for target address generation patterns. For example, Wartell et al. [45] scan the binary to identify call instructions and perform simple data-flow analysis to identify instructions that use the generated address in an arithmetic computation.

3.1.8 Security Analysis

Evading Total-CFI In this work, we address the attacks that arise due to control flow violations. Most attacks in the wild are control hijacking attacks, where attacker executes malicious payload by diverting control flow. However, there exist data only attacks [46] that do not hijack control flow (e.g., bad system configuration resulting in unintended privilege escalation). Such attacks are out of our scope. That said, works in the past [47–49] have focussed on addressing data integrity concerns. There also exist techniques based on

dynamic taint analysis [50–52] wherein, input data is marked as tainted and tracked through memory to ensure that they do not end up in security critical data structures. `Total-CFI` relies on integrity of kernel data to guarantee the correctness of perceived events in the guest kernel. Since `Total-CFI` retrieves the data directly from the guest OS kernel data structures, attacks that tamper with the kernel data will mislead `Total-CFI`.

Furthermore, non-control flow side channel attacks, data attacks [53, 54], physical attacks and attacks that target the VMM are also out of `Total-CFI`'s scope. Attacks against the VMM have been demonstrated in the past [55].

Exploits within whitelist `Total-CFI` treats the entries in the whitelist as legal entries for indirect branch operations. Therefore, all the function entry points (such as libc functions) belong to the whitelist. This gives rise to a possibility for an attacker to craft an attack such that the jump/call target is an entry within the whitelist. Currently, `Total-CFI` is vulnerable to such `jump-or-call-to-libc` type of attacks. Note that return-to-libc will be captured by `Total-CFI` due to the violation in the shadow call stack.

3.1.9 Summary

`Total-CFI` [42] recovers the function-level semantics, specifically function entry points, in order to improve the precision of CFI policy. In comparison with DEP-only defense, `Total-CFI` provides much stronger protection, and can detect all tested exploits. Moreover, due to a system-wide nature of deployment, `Total-CFI` can provide *whole system* CFI protection.

3.2 Precise CFI Model through Recovery of C++ Semantics

While `Total-CFI` and other coarse-grained CFI solutions [30] [40] have significantly reduced the attack surface, recent efforts by Göktaş et al. [11] and Carlini [10] have demonstrated that coarse-grained CFI solutions are too permissive, and can be bypassed by reusing large gadgets whose starting addresses are allowed by coarse-grained solutions. We argue that the primary reason for such permissiveness is the lack of high-level program semantics that introduce certain mandates on the control flow. For example, given a class inheritance, target of a virtual function dispatch in C++ must be a virtual function that the dispatching object is allowed to invoke. Similarly, target of an exception dispatch must be one of the legitimate exception handlers. Accounting for control flow restrictions imposed at higher levels of semantics improves the precision of CFI.

In this work, we recover C++-level semantics to generate more precise CFI policies for dynamic dispatches in C++ binaries. We set our focus on C++ binaries because, due to its object-oriented programming paradigm and high efficiency as compared to other object-oriented languages like Java, it is prevalent in many complex software programs. To support polymorphism, C++ employs a dynamic dispatch mechanism. Dynamic dispatches are predominant in C++ binaries and are executed using an indirect `call` instruction. For instance, in a large C++ binary like libmozjs.so (Firefox's Spidermonkey Javascript engine), 84.6% indirect function calls are dynamic dispatches. For a given C++ binary, we aim to construct sound and precise CFI policy for dynamic dispatches in order to reduce the space for code-reuse attacks.

Constructing a strict CFI policy directly from C++ binaries is a challenging task. A strict CFI policy should not miss any legitimate virtual call targets to ensure zero false alarms, and should exclude as many impossible virtual call targets as possible to reduce the attack space. In order to protect real-world binaries, all these need to be accomplished under the assumption that only the binary code (without any symbol or debug information) is available. In order to construct a strict CFI policy for virtual calls, we need to reliably rebuild certain C++-level semantics that persist in the stripped C++ binaries, particularly VTables and virtual callsites. Based on the extracted VTables and callsites, we can construct a basic CFI policy and further refine it. As a key contribution, we demonstrate that CFI policies with increased precision can be constructed by recovering C++-level semantics. While the refined policies may not completely eliminate code-reuse attacks, by reducing the number of available gadgets, it makes attacks harder to execute. A complete version of the paper is available [56].

3.2.1 Polymorphism in C++ Binary

In C++, functions declared with keyword "virtual" are termed "virtual functions" [57] and their invocation is termed "virtual call" (or vcall). Virtual functions are in the heart of polymorphism, which allows a derived class to override methods in its base class. When a virtual function that is overridden in a derived class is invoked on an object, the invoked function depends on the object's type at runtime. Modern compilers – e.g., Microsoft Visual C++ (MSVC) and GNU g++ – achieve this resolution using a "Virtual Function Table" or VTable, a table that contains an array of "virtual function pointers" (vfptr) –

pointers to virtual functions. Itanium [58] and MSVC [59] are two of the most popular C++ ABIs that dictate the implementation of various C++ language semantics.

Objects and VTables An object contains various instance variables along with the vptr. Due to its frequent use, modern compilers place vptr as the first entry within the object. The vptr is followed by the member variables of the class. The location in the VTable where the vptr points to is called the "address point". The first vfptr in the VTable is stored at the address point.

In addition to an array of virtual function pointers, a VTable also holds optional information at negative offsets from address point. Optional information includes Run Time Type Information (RTTI) and various offset fields required to adjust the *this* pointer at runtime. We refer readers to [58] for more information on RTTI and various offset fields. During compilation, all the VTables used by a module are placed in a read-only section of the executable. Furthermore, a hidden field called "virtual table pointer" (vptr) – a pointer to the VTable – is inserted into objects of classes that either directly define virtual functions or inherit from classes that define virtual functions. Under normal circumstances, the vptr is typically initialized during construction of the object.

Virtual Call Dispatch Irrespective of the compiler optimizations and the ABI, a virtual call dispatch comprises of the following 5 steps:

GetVT Dereference the vptr of the object (*this* pointer) to obtain the VTable.

GetVF Dereference (VTable + *offset*) to retrieve the vfptr to the method being invoked.

SetArg Set the arguments to the function on the stack or in the registers depending on the calling convention.

SetThis Set the implicit *this* pointer either on stack or in `ecx` register depending on the calling convention.

CallVF Invoke the vfptr using an indirect `call` instruction.

GetVT, GetVF, SetThis and CallVF are required steps in all vcalls, whereas depending on if the callee function accepts arguments or not, SetArg is optional. Though there is no restriction with respect to relative ordering of the steps followed, some steps are implicitly dependent on others (e.g., GetVF must occur after GetVT).

3.2.2 Problem Statement, Assumptions and Scope

Problem Statement Given a C++ binary, we aim to construct a CFI policy to protect its virtual function calls (or dynamic dispatches). Specifically, for each virtual callsite in the binary, we need to collect a whitelist of legitimate call targets. If a call target beyond the whitelist is observed during the execution of the C++ binary, we treat it as a violation against our CFI policy and stop the program execution.

More formally, this CFI policy can be considered as a function:

$$\mathcal{P} = \mathcal{C} \rightarrow 2^{\mathcal{F}},$$

where $\mathcal{C}$ denotes all virtual function call sites and $\mathcal{F}$ all legitimate call targets inside the given C++ binary. Therefore, as a power set of $\mathcal{F}$, $2^{\mathcal{F}}$ denotes a space of all subsets of $\mathcal{F}$.

Furthermore, we define callsite $c \in \mathcal{C}$ to be a 2-tuple, c = *(displacement, offset)* with displacement from the base of the binary to the callsite and the VTable offset at the callsite.

A good CFI policy must be *sound* and as *precise* as possible. The existing binary-only CFI solutions (e.g., BinCFI, CCFIR, etc.) ensure soundness, but are imprecise, and therefore expose considerable attack space to sophisticated code-reuse attacks [11]. Therefore, to provide strong protection for virtual function calls in C++ binaries, our CFI policy must be sound, and at the same time, be more precise than the existing binary-only CFI protections.

To measure the precision, we can use source-code based solutions as reference systems. With source code, these solutions can precisely identify the virtual dispatch callsites and the class inheritance hierarchy within the program. Then, at each callsite, they insert checks to ensure that (1) the VTable used to make the call is compatible with the type of the object making the call [6] or (2) the call target belongs to a set of polymorphic functions extracted from the inheritance tree for the type of object making the call [4].

Assumptions and Scope Since we target COTS C++ binaries, we must assume that none of source code, symbol information, debugging information, RTTI, etc. is available. We must also deal with challenges arising due to compiler optimizations that blur and remove C++ semantic information during compilation. In other words, we must rely on strong C++ semantics that are dictated by C++ ABIs and persist during the process of code compilation and optimization. Due to the reliance on standard ABIs, we only target C++ binaries that are compiled using standard C++ compilers (e.g., MSVC and GNU g++). Custom compilers that do not adhere to Itanium and MSVC ABIs are out of scope.

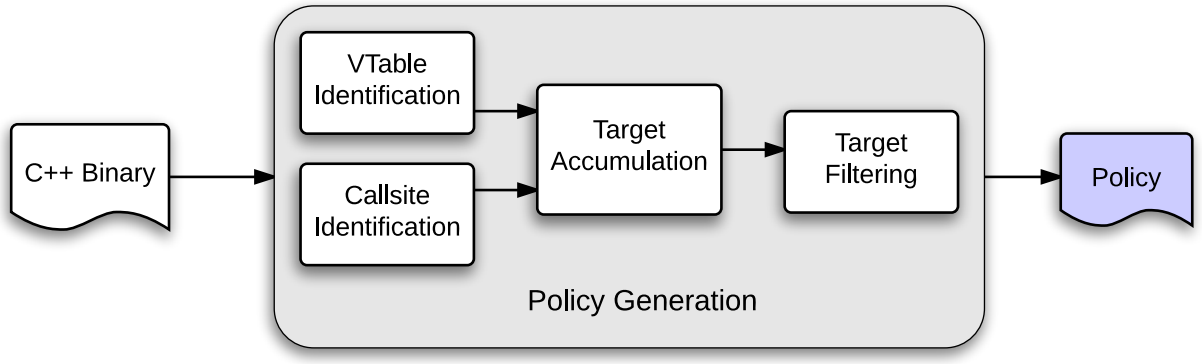

Fig. 3.4.: Overview of `vfGuard`

Moreover, since our goal is to protect benign C++ binaries, code obfuscation techniques that deliberately attempt to evade and confuse our defense are also out of our scope.

Furthermore, our goal is to protect virtual function calls and their manifestations through indirect `call` instructions in the binary. We do not aim to protect indirect `jmp` or `ret` instructions. However, we aim to provide a solution orthogonal to existing solutions (e.g., shadow call stack [60], coarse-grained CFI [30, 40]) so as to provide a more complete and accurate CFI.

3.2.3 Approach Overview

In order to tackle the problem stated in Section 3.2.2, we must leverage the C++ ABIs to recover strong C++ semantics that persist in a given C++ binary. First of all, we need to accurately discover virtual callsites $\mathcal{C}$ in the binary. Then, we need to identify all the virtual function entry points, which form the legitimate call targets $\mathcal{F}$. Because all functions in $\mathcal{F}$ are polymorphic (virtual), and must exist in VTables, we must identify all the VTables in the binary. After having identified virtual callsites and VTables, we can

construct a basic policy such that, for each callsite the legitimate targets include all the functions in the VTables at the given offset. This basic policy is already more precise than `Total-CFI` and other state-of-the-art CFI defenses – CCFIR and BinCFI. To further improve the policy precision, we propose two additional filters to reduce this set of legitimate targets.

Figure 3.4 presents the overview of our solution. We have implemented our solution in a tool called `vfGuard`. Given a C++ binary, `vfGuard` will extract virtual callsites in the "Callsite Identification" component, and VTables in the "VTable Identification" component. With the extracted callsites and the VTables, the "Target Accumulation" component accumulates all the functions from the VTables that individual callsites can target. Finally, "Target Filtering" filters the targets to obtain a more precise policy. The generated policy can be enforced using one of several ways discussed in Chapter 4.

3.2.4 Callsite Identification

Challenges The key challenge in identifying the callsites is to differentiate legitimate callsites from other indirect call instructions. C++ binary is often mixed with non-C++ code components written in C, assembly language, etc. These code components could be included from a dependent library or could be injected by the compiler for exception handling, runtime binding, etc. While the steps enumerated in Section 3.2.1 provide a good starting point to look for callsites within a binary, they present some hard challenges.

- Some steps are independent of the other and therefore follow no strict ordering – e.g., SetArg and SetThis may occur before GetVT and GetVF.

- GetVT through CallVF may span multiple basic blocks – e.g., when two or more virtual calls are dispatched in multiple branches on the same *this* pointer, a compiler may move some steps to a basic block that dominates the blocks that performs the calls.

- Some steps may be implicit. For example, in functions that employ `thiscall` convention, an incoming object pointer may be retained in the `ecx` register throughout the function thereby eliminating the need for an explicit SetThis for virtual calls on the same object.

- GetVF resembles a simple dereference if offset is 0. This bears close resemblance to a double dereference of a function pointer in C code. To ensure soundness in our policy generation, we cannot afford to include false virtual callsites.

To address all the above challenges, we take a principled approach and perform static and flow-sensitive intra-procedural data flow analysis on an intermediate representation of the binary.

Intra-Procedural Static Analysis Due to the complex nature of x86 binaries and the complexities involved in recovering the callsites, a simple scanning-based approach for callsite identification is insufficient. `vfGuard` first identifies all the candidate functions that could host callsites by identifying functions in the binary that contain at least 1 indirect call instruction. Each identified function is subjected to static intra-procedural analysis to identify legitimate virtual callsites and VTable offsets at such callsites. In order to perform

$function$	::=	$(stmt)*$
$stmt$	::=	$var ::= exp \mid exp ::= exp \mid$ `goto` exp
		$\mid$ `call` $exp \mid$ `return`
		$\mid$ `if` var `then` $stmt$
exp	::=	$exp \Diamond_b exp \mid \Diamond_u exp \mid var$
$\Diamond_b$	::=	$=, +, -, *, /, ...$
$\Diamond_u$	::=	$deref, -, \sim$
var	::=	$\tau_{reg} \mid \tau_{val}$
τ_{reg}	::=	`reg1_t|reg8_t|reg16_t|reg32_t`
τ_{val}	::=	$\{Integer\}$

Table 3.1: Intermediate Language used by `vfGuard`

the data flow analysis, we modified an open source `C` decompiler [61]. Below, we present the different steps in our analysis.

IR Transformation and SSA Form: x86 instruction set is large, and instructions often have complex semantics. To aid in analysis and focus on the data flow, we make use of a simple, yet intuitive intermediate language as shown in Table 3.1. The IR is simple enough to precisely capture the flow of data within a function without introducing unnecessary overhead. Each function is first broken down into basic blocks and a control-flow graph (CFG) is generated. Then, starting from the function entry point, the basic blocks are traversed in a depth-first fashion and each assembly instruction is converted into one or more IR statements. A statement comprises of expressions, which at any point is a symbolic representation of data within a variable. A special unary operator called $deref$ represents the dereference operation. `goto, call` and `return` instructions are retained with similar semantic interpretations as their x86 counterparts. Edges between basic blocks in the CFG is captured using `goto`.

In its current form, `vfGuard` supports registers up to 32 bits in size, however, the technique itself is flexible and can be easily extended to support 64-bit registers. Moreover, note that the ABIs are not restricted to any particular hardware architecture. Performing analysis on the IR facilitates our solution to be readily ported to protect C++ binaries on other architectures (e.g., ARM) by simply translating the instructions to IR.

Furthermore, we convert each IR statement into Single Static Assignment (SSA) [25, 62] form, which has some unique advantages. IR in SSA form readily provides the *def-use* and the *use-def* chains for various variables and expressions in the IR.

Def-Use Propagation: The definition of each SSA variable and list of statements that use them constitutes the Def-Use chains [25]. `vfGuard` recursively propagates the definitions into uses until all the statements are comprised of entry point definitions (i.e., function arguments, input registers and globals). Due to flow-sensitive nature of our analysis, it is possible that upon propagation, we end up with multiple expressions for each SSA variable, and each expression represents a particular code path. For example, consider the code snippet:

```
...
1.  A *pa; A a; C c;
2.  if (x == 0)
3.     pa = &a
4.  else
5.     pa = &c
6.  pa->vAtest(0);
```

...

At line 6, depending on the value of `x`, the vfptr corresponding to `vAtest` could either be `&(&(&c)+0x14)` or `&(&(&a)+0x14)`. Assuming `stdcall` convention, per step SetThis, the implicit object pointer could either be `&c` or `&a`. Precise data flow analysis should capture both possibilities, and for each case ensure the existence of a corresponding *this* pointer assignment on the stack. For such cases, `vfGuard` creates multiple copies of the statement – one for each propagated expression.

Subsequently, each definition is recursively propagated to the uses until a fixed point is reached. At each instance of propagation, the resulting expression is simplified through constant propagation. For example, $deref((ecx_0 + c_1) + c_2)$ becomes $deref(ecx_0 + c_3)$ where $c_3 = c_1 + c_2$.

Address	**Instruction**	**IR-SSA form**	**After Propagation and Constant Folding**
0x798	`push` ebp	$deref(esp_0) = ebp_0$ $esp_1 = esp_0 - 4$	$deref(esp_0) = ebp_0$ $esp_1 = esp_0 - 4$
0x799	`mov` ebp, esp	$ebp_1 = esp_1$	$ebp_1 = esp_0 - 4$
0x79b	`sub` $esp, 0x18h$	$esp_2 = esp_1 - 0x18$	$esp_2 = esp_0 - 0x1C$
0x79e	`mov` $eax, [ebp + 8]$	$eax_0 = deref(ebp_1 + 8)$	$eax_0 = deref(esp_0 + 4)$
0x7a1	`mov` $eax, [eax]$	$eax_1 = deref(eax_0)$	$eax_1 = deref(deref(esp_0 + 4))$
0x7a3	`add` $eax, 8$	$eax_2 = eax_1 + 8$	$eax_2 = deref(deref(esp_0 + 4)) + 8$
0x7a6	`mov` $eax, [eax]$	$eax_3 = deref(eax_2)$	$eax_3 = deref(deref(deref(esp_0 + 4)) + 8)$
0x7a8	`mov` $edx, [ebp + 8]$	$edx_0 = deref(ebp_1 + 8)$	$edx_0 = deref(esp_0 + 4)$
0x7ab	`mov` $[esp], edx$	$deref(esp_2) = edx_0$	$deref(esp_2) = deref(esp_0 + 4)$
0x7ae	`call` eax	`call` eax_3	`call` $deref(deref(deref(esp_0 + 4)) + 8)$

Table 3.2: Static information flow analysis to identify a callsite

Callsite Labeling As per the steps involved in dynamic dispatch, we need to capture GetVT through CallVF using static data flow analysis. More specifically, for each indirect call, we compute expressions for the call target and expressions for *this* pointer passed to

the target function. Note that due to flow-sensitive data flow analysis, we may end up having multiple expressions for each statement or variable.

For a virtual callsite, after def-use propagation, its call instruction must be in one of the two forms:

$$\texttt{call}\ deref(deref(exp) + \tau_{val}) \tag{3.1}$$

or

$$\texttt{call}\ deref(deref(exp)) \tag{3.2}$$

In the first form, *exp* as an expression refers to the vptr within an C++ object and τ_{val} as a constant integer holds the VTable offset. When a virtual callsite invokes a virtual function at offset 0 within the VTable, the call instruction will appear in the second form, which is a double dereference of vptr. Here, τ_{val} is the byte offset within the VTable and must be divisible by the pointer size. Therefore, if τ_{val} is not divisible by 4, the callsite is discarded.

Next, we need to compute an expression for *this* pointer at the callsite. *this* pointer can be either passed through `ecx` in `thiscall` or pushed onto the stack as the first argument in `stdcall` conventions. Expression for *this* pointer must be identical to the *exp* within the form (1) or (2).

Table 3.2 presents a concrete example. At 0x7ae, after propagation and simplification, the call instruction matches with form (1) and we determine the expression for *this* pointer to be $deref(esp_0 + 4)$ and VTable offset as 8. Then at 0x7ab, we determine that the first argument pushed on the stack is also $deref(esp_0 + 4)$. Now, we are certain that this callsite is indeed a virtual callsite, and it uses `stdcall` calling convention.

Effect of Inheritance on Virtual Callsites: It is worth noting that our technique is independent of the inheritance structure and works not only for single inheritance, but also multiple and virtual inheritances. This is because the compiler adjusts *this* pointer at the callsite to point to appropriate base object *before* the virtual function call is invoked. Therefore, while the expression for *this* pointer may vary, it must be of the form (1) or (2) above. Our method aims to resolve *this* pointer directly for each callsite, and thus can deal with all these cases.

3.2.5 VTable Identification

Challenges Reconstructing precise inheritance tree from the binary is ideal but hard. For instance, Dewey et al. [63] locate the constructors in the program by tracking the VTable initializations. In commercial software, constructors are often inlined, therefore such an approach would not yield a complete set of VTables that we seek. Other approaches use heuristics that are not only dependent on debug information, but also tailored for specific compilers like MSVC (e.g., IDA VTBL plugin [64]).

We propose an ABI-centric algorithm that can effectively recover *all* the VTables in a binary in both MSVC and Itanium ABIs. We make the following key observations:

Ob1: VTables are present in the read-only sections of the binary.

Ob2: Offset of vfptr within a VTable is a constant and is statically determinable at the invocation callsite.

Ob3: Since the caller must pass the *this* pointer, any two polymorphic functions must adhere to the same calling convention.

Based on **Ob1**, we scan the read-only sections in the binary to identify the VTables. The VTables that adhere to Itanium ABI contain mandatory fields along with the array of vfpts. The mandatory fields make locating of VTables in the binary relatively easier when compared to MSVC ABI. However, VTables generated by the Microsoft Visual Studio compiler (MSVC ABI) are often grouped together with no apparent "gap" between them. This poses a challenge to accurately identify VTable boundaries.

Furthermore, according to **Ob2**, we first scan the code and data sections and identify all the "immediate" values. Then, we check each value for a valid VTable address point. A valid VTable contains an array of one or more vfptrs starting from the address point. It is possible that our algorithm identifies non-VTables – e.g., function tables that resemble VTables – as genuine VTables. We err on the safe side, because including a few false VTables does not compromise the policy soundness and only reduces precision to a certain degree. A detailed algorithm for VTable identification is presented below.

VTable Scanning Algorithm The algorithm used to identify VTables is presented in Algorithm 1. It comprises of two functions. "ScanVTables" takes a binary as input and returns a list of all the VTables $\mathcal{V}$ in the binary. Each instruction in the code sections and each address in the data sections of the binary are scanned for immediate values. If an immediate value that belongs to a read-only section of the binary is encountered, it is checked for validity using "getVTable" and $\mathcal{V}$ is updated accordingly. "getVTable" checks and returns the VTable at a given address. Starting from the address, it accumulates

entries as valid vfptrs as long as they point to a valid instruction boundary within the code region. Note that not all valid vfptrs may point to a function entry point. For instance, in case of "pure virtual" functions, the vfptr points to a compiler generated stub that jumps to a predefined location. In fact the MSVC compiler introduces stubs consisting of a single return instruction to implement empty functions. To be conservative, `vfGuard` allows a vfptr to point to any valid instruction in the code segments. Upon failure, it returns the accumulated list of vfptrs as the VTable entries. If no valid vfptrs are found, an empty set – signifying invalid VTable address point – is returned.

Furthermore, the following restriction is imposed on Itanium ABI: A valid VTable in the Itanium ABI must have valid RTTI and "OffsetToTop" fields at negative offsets from the address point. The RTTI field is either 0 or points to a valid RTTI structure. Similarly, "OffsetToTop" must also have a sane value. A value -0xffffff $\leq$ *offset* $\leq$ 0xffffff, which corresponds to an offset of 10M within an object, was empirically found to be sufficient. Depending on the specific classes and inheritance, fields like "vbaseOffset" and "vcallOffset" may be present in the VTable. To be conservative, we do not rely on such optional fields. However with stronger analysis and object layout recovery, these restrictions can be leveraged for more precise VTable discovery.

While `vfGuard` may identify some false VTables as legitimate, its conservative approach does not miss a legitimate VTable, which is a core requirement to avoid false positives during enforcement. Moreover, Algorithm 1 is not very effective at detecting end points of the VTables in the binary. Pruning the VTables based on neighboring VTable start addresses could lead to unsound policies if the neighboring VTables are not legitimate.

Algorithm 1 Algorithm to scan for VTables.

```
1: procedure getVTable(Addr)
2:     V_methods ← ∅
3:     if (ABI_Itanium and isMandatoryFieldsValid(Addr)) or ABI_MSVC then
4:         M ← [Addr]
5:         while isValidAddrInCode(M) do
6:             V_methods ← V_methods ∪ M
7:             Addr ← Addr + size(PTR)
8:             M ← [Addr]
9:         end while
10:    end if
11:    return V_methods
12: end procedure
13:
14: procedure ScanVTables(Bin)
15:    V ← ∅
16:    for each Insn ∈ Bin.code do
17:        if Insn contains ImmediateVal then
18:            C ← immValAt(Insn)
19:            if C ∈ Section_RO and getVTable(C) ≠ ∅ then
20:                V ← V ∪ C
21:            end if
22:        end if
23:    end for
24:    for each Addr ∈ Bin.data do
25:        if [Addr] ∈ Section_RO and getVTable([Addr]) ≠ ∅ then
26:            V ← V ∪ [Addr]
27:        end if
28:    end for
29:    return V
30: end procedure
```

While our approach reduces precision, it keeps the policy sound. Our algorithm terminates a VTable when the vfptr is an invalid code pointer.

While Itanium ABI provides strong signatures for VTables due to mandatory offsets, MSVC ABI does not. In theory, any pointer to code can be classified as a VTable under MSVC ABI. In practice however, we found that legitimate VTables contain at least 2 or more entries. Therefore, under MSVC ABI, we consider VTables only if they contain at least 2 entries.

3.2.6 Target Accumulation and Filtering

All the vfptrs and *thunks* within all the VTables together form a universal set for virtual call targets. A naive policy will include *all* vfptrs as valid targets for each callsite. For large binaries, such a policy would contain 1000s of targets per callsite. While still more precise than existing defenses, it would still expose a large attack space. We leverage the offset information at the callsite to obtain a more precise policy.

Given an offset at a callsite, during "Target Accumulation", we obtain a basic policy for each callsite that encompasses all the vfptrs (and *thunks*) at the given offset in all the VTables in which the offset is valid. Additionally, we apply two filters to further improve the policy precision. First, we note that target vfptrs for a callsite that is invoked on the same object pointer as the host function must belong to the same VTables as the host function. With this, we apply our first filter, called "Nested Call Filter". Furthermore, from **Ob 3**, the calling convention that is presumed at the callsite and the calling convention adhered to by the target function must be compatible. Accordingly, we apply the second filter called "Calling Convention Filter".

We considered several other filters, but did not adopt them for various reasons. To name a few, we could infer the number of arguments accepted by each function and require it to match the number of arguments passed at the callsite; we could perform inter-procedural data flow analysis to keep track of *this* pointers; and we could perform type inference on function parameters and bind type compatible functions together. However, at a binary level, such analyses are imprecise and incomplete. A function may not always use all the arguments declared in source code, and thus we may not reliably

obtain the argument information in the binary. Inter-procedural data flow analysis and type inference are computationally expensive, and by far not practical for large binaries. We will investigate more advanced filters as future work.

Basic Policy Based on the identified callsites and the VTables V, `vfGuard` generates a basic policy. For a given callsite c with byte offset o, we define index k to be the index within the VTable that the byte offset corresponds to (i.e., $k = o/4$ for 4-bytes wide pointers). The legitimate call targets of c must belong to a subset of all the functions at index k within in all the VTables that contain at least $(k + 1)$ vfptrs. Here we assume VTables to be zero-based arrays of vfptrs. That is:

$$Targets = \{V_i[k] \mid V_i \in \mathcal{V},\ |V_i| > k\},$$

where V_i is the VTable address point. $|V_i|$ is the number of vfptrs in the VTable at V_i.

Nested Virtual Call Filter In some cases, *this* pointer used to invoke a virtual function is later used to make one or more virtual calls within the function body. We refer to such virtual calls as "Nested Virtual Calls". `vfGuard` can generate a more precise policy for nested virtual calls.

```
1. class M { virtual void vMfoo() {
2.     this->vFn(); //or vFn();
3.   }
4. };
```

In the above example, `vFn` is a virtual function that is invoked on the same *this* pointer as its host function `M::vMfoo`, which is also a virtual function. Underneath, the binary implementation reuses the VTable used to invoke `M::vMfoo` to retrieve the vfptr (or *thunk*) pertaining to `vFn`. That is, between the nested virtual calls and the host virtual function, the vptr acts an invariant. Therefore at the nested callsite, target `vFn` must belong to a VTable to which `M::vfoo` also belongs.

Given a virtual callsite with vfptr index k and host virtual function f, we can derive a more precise policy for each nested virtual callsite within f:

$$Targets = \{V_i[k] \mid V_i \in \mathcal{V}, f \in V_i\}$$

Nested virtual callsites can be easily identified using our intra-procedural data flow analysis. First, we check whether the *this* pointer at the given callsite is in fact the *this* pointer for the host function. That is, the expression for *this* pointer at that callsite should be ecx_0 for `thiscall` calling convention or $esp_0 + 4$ for `stdcall` calling convention. Next, we ensure that the host function is virtual. That is, there must exist at least 1 VTable to which the host function belongs. Finally, the filtered targets are identified using the equation above.

Note that the filter is applicable only in cases where host function is also virtual. If the filter is inapplicable, `vfGuard` defaults to basic policy.

Calling-Convention based Filtering We filter the target list to be compatible with the calling convention followed at the callsite. At the callsite, the register that is utilized to

pass the implicit *this* pointer (CallVF) reveals the calling convention that the callee function adheres to.

First, for each of the callsite target functions in the policy, we identify the calling convention the function adheres to. Next, for each callsite, we check if there is a mismatch between the convention at the callsite and the target, if so, we remove such conflicting targets from the list. If we are unable to identify the calling convention of the callee – which is possible if the callee does not use the implicit *this* pointer, we take a conservative approach and retain the target.

Incomplete Argument Utilization Conceptually, all polymorphs of a function must accept the same number of arguments. So, one potential filter could be to check if a function accepts the same number of arguments that are passed at the callsite. If not, the mismatching functions can be removed from potential targets for the callsite. However, in the binary, we only see the number of arguments *used* by a function and not the number of arguments it *can* accept. Therefore, argument count is *not* feasible as a filter.

3.2.7 Discussion

How to better identify VTable end points The strictness of the policy or the attack space depends on the number of call-targets per callsite. Ideally, we want this number to be as close to the ideal case as possible. However, inaccurate VTable end-points – specially in MSVC ABI – result in inclusion of incorrect vfptrs into the policy.

Consider the layout in Figure 3.5. V_A and V_B are 2 VTables under MSVC ABI that are contiguously allocated, where V_A is comprised of entries f_{1-4} and V_B of entries f_{5-6}. I_{1-3}

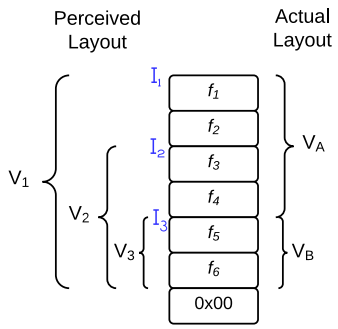

Fig. 3.5.: Actual and perceived VTable layouts under MSVC ABI.

are addresses within the function pointer array that manifest within the binary as immediate values, where I_1 and I_3 are VTable address points and I_2 is noise. Per Algorithm 1, `vfGuard` identifies 3 VTables V_{1-3} with a total of 6+4+2=12 entries. Accordingly (due to V_2) f_3, f_4, f_5 and f_6 are incorrectly included in the policy for offsets 0, 1, 2 and 3 respectively, thereby compromising precision over soundness. Similarly, f_5 and f_6 are allowed as legitimate targets for offsets 4 and 5 respectively.

One solution to such a problem would be to prune VTables based on the start addresses of succeeding VTables in the memory. However, such a solution must have no false positives in VTable start-addresses. In Figure 3.5, since I_2 is an incorrect VTable address point, f_{3-4} would be incorrectly excluded from V_1 thereby leading to false positives during enforcement. Another solution could leverage more restrictions from the ABI and language semantics to better demarcate VTables. For example, colocated functions at a given offset must be compatible with each other with respect to types of arguments accepted and type of value returned.

Virtual-dispatch-like C calls Our virtual callsite identification has captured all required steps for a virtual dispatch according to C++ ABI specifications, but it is still possible that some functions in the C code could resemble a legitimate C++ virtual function dispatch. For example:

```
struct B{void *ptr; void (*fn) (struct A*);};
struct A{struct B *pb;};
void foo(struct A* pa) {
pa->pb->fn(pa);
}
```

In the binary, the above C statement resembles a C++ virtual call dispatch. `pa` being passed as an argument satisfies SetThis and could be perceived as a callsite. Commercial compilers tend to follow a finite number of code patterns during a virtual call invocation. A potential solution could classify all the callsites based on code patterns used to perform the dispatch and look for abnormalities. For example, to dispatch virtual calls in mshtml.dll, the compiler typically invokes virtual calls using a call instruction of the form, "`call` [*reg + offset*]" where as, g++ produces code that performs, "`add` *reg, offset*; `call` *reg*". While this is not a standard, a compiler tends to use similar code fragments to dispatch virtual calls within a given module. Since a sound policy must prevent false callsites, one can filter the potential incorrect callsites by looking for persistent virtual dispatch code fragments.

3.2.8 Summary

`vfGuard` recovers C++-level semantics, particularly virtual callsites and VTable information from C++ binaries to generate precise CFI policy. It offers more stringent protections for virtual function dispatches – over 95% more precise when compared to `Total-CFI` and other state-of-the-art binary-level defenses.

3.3 Attack Space Reduction through Stack-Pointer Integrity (SPI)

While `Total-CFI` increased precision of CFI policies, and `vfGuard` improved it further by recovering C++-level semantics, the lack of *all* pertinent high-level semantics lead to an theoretical attack surface. For example, with access to source code, SafeDispatch [4] and VTV [6] can recover precise C++ class hierarchy and therefore generate ideal policies for virtual function dispatches.

A recent attack called Counterfeit Object Oriented Programming (COOP) [65] takes advantage of attack surface made available by C++-level defenses including `vfGuard`. While `vfGuard` is most resilient to COOP when compared to other binary-level defenses, it does suffer from some limitations. Over time, attacks and defenses against code-reuse attacks have led to a cat and mouse game that has prompted to rethink defense against code-reuse attacks.

ROP, a particular type of code-reuse attack, is so popular and mature as an attack mechanism that there exist tools to automatically extract gadgets [8] from programs and compile them [9] (i.e., chain them) to implement program logic.

A key component of 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. Our defense stems from the observation that during ROP, each gadget behaves like an instruction with complex semantics, and the stack pointer performs the role of a program counter. Therefore, traditional CFI, which imposes integrity restrictions on the program counter under the regular execution domain, transforms into integrity restrictions on the stack-pointer in the ROP domain. Fundamentally, SPI as a property requires that: (1) at any point during execution, the stack pointer remains within the stack region of the currently executing thread, and (2) stack pointer is conserved across function execution. That is, the size of the stack frame allocated to a function for execution is equal to the size of the stack frame deallocated after the execution of the function. During the stack-pivot operation of an ROP attack, at least one of the two requirements is violated. Based on location of the payload, we divide SPI into two categories: Coarse-Grained SPI, which defends against attacks where the ROP payload is on non-stack segments (e.g., heap), and Fine-Grained SPI, which defends against attacks where the ROP payload is on the stack.

SPI has several advantages over prior binary-level ROP defenses:

1. SPI is a non-control-flow approach and makes no assumptions regarding the size or instruction semantics of gadgets. In fact, SPI is oblivious to the concept of a "gadget", and operates at the instruction level.

2. SPI is impervious to ASLR. Threat model addressed by SPI allows for ASLR to be turned off and yet, defend against ROP.

3. SPI allows for incremental deployment. That is, only specific modules can be protected, and the protected modules can inter-operate with unprotected modules.

4. Finally, coarse-grained SPI – which is sufficient to protect against all heap-based ROP attacks – offers very low overhead.

3.3.1 Motivation

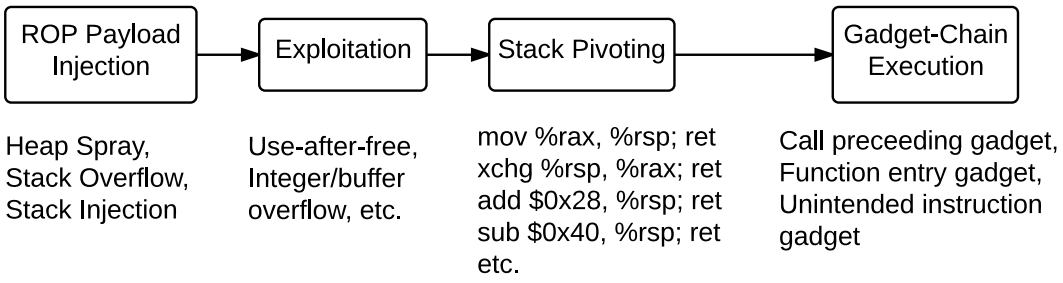

Fig. 3.6.: Steps involved in executing a typical ROP attack.

Code-Reuse Attacks A schematic overview of steps involved in a ROP attack is presented in Figure 3.6. Jump-Oriented Programming (JOP) [66], a variant of ROP uses a `pop` followed by an indirect `jmp` 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 JOP and other known code-reuse attacks. 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 that the attack executes. Note that ROP payload is different from the malicious executable payload that is commonly executed after DEP is bypassed.

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. The payload can also be injected into the stack region, where the attacker often misuses stack variables, or in the case of stack overflow, overflows the stack to store the ROP payload. 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, etc.). At the end of this step, the attacker controls the program counter. She may also control certain registers depending on the nature of the attack.

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 instruction (typically `ret` or `pop` *reg* followed by `jmp` *reg*) at the end of the stack pivot gadget triggers the execution of the chain of gadgets directed by the payload.

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 location of the payload and 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; etc.`). 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`, etc. are absolute SP-updates.
  - *Relative SP-update:* These instructions alter the stack pointer by a fixed offset (e.g., `add esp, 0x10; sub esp, 0x10; etc.`).
- **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.

Typically, stack-pivot requires an explicit SP-update instruction. On one hand, because absolute SP-update instructions can load an arbitrary value into the stack pointer, they are a popular choice for stack pivot. However, they typically require that the attacker controls a register and loads it with the address of the payload, which depending on the nature of the exploit, may or may not be possible. On the other hand, relative SP-update instructions are useful to move the stack pointer by fixed relative offsets, and therefore are useful when the payload is within the stack region. Implicit SP-update instructions are least capable because they can only move the stack pointer by small offsets. Most practical

attacks store payload on the heap, and utilize an absolute SP-update instruction for pivoting. However, a determined attacker can, and will resort to using relative SP-update instruction if defense against abuse of absolute SP-update instructions for pivoting becomes prevalent. In this paper, we focus on eliminating stack pivot operations that utilize explicit SP-update instructions. Whereas implicit SP-updates can in principle be used for stack pivot, we did not find any in practice. We elaborate on implicit SP-updates in Section 5.4.

Stack Pivot within Stack Region: When the payload is located within the stack region, stack pivot moves the stack pointer either along the direction or against the direction of stack growth. We define *Stack-Forward Pivot* to signify movement of stack pointer along the direction of stack growth, and *Stack-Backward Pivot* to signify movement of stack pointer against the direction of stack growth. Since stack pointer always points to the top of active (or live) stack region, stack-forward pivot implies that the payload is located in the stale (or dead) region of the stack (Figure 3.7(b)). Assuming that stack grows from higher to lower address, stack-forward pivoting is usually accomplished by subtracting an offset from the existing value of stack pointer. Stack-backward pivoting (Figure 3.7(a)) moves the stack pointer into the active region of the current function's or a callee function's stack frame by adding an offset to the stack pointer.

As an example, In Figure 3.7(a), the attacker performs stack-backward pivoting. She 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. In Figure 3.7(b), the attacker performs stack-forward pivoting by pivoting 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.

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 same 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.

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` and invoke it when the local variables for a function exceed 4K and 8K for 32 and 64 bit architectures respectively. `_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.

3.3.2 SPI – Overview

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 program counter), pivoting the stack violates the integrity of stack pointer. With this in mind, we present Stack-Pointer Integrity (SPI), a program property that mandates two runtime invariants:

- **Stack Localization (P1):** At any point during execution of a program, stack frame (or stack pointer) of the currently executing function exists within the stack region of the currently executing thread.
- **Stack Conservation (P2):** The size of the stack frame allocated before the execution of a function is equal to the size of the stack frame deallocated after execution of the function.

We observe that during a ROP attack, depending on the location of payload, stack pivoting violates at least one of **P1** or **P2**. We divide SPI into Coarse- and Fine-Grained

SPI. An overview of defense based on payload location is presented in Table 3.3. Coarse-Grained SPI enforces *Stack Localization* in order to defeat all attacks that pivot to payloads located outside the stack region (e.g., heap), and Fine-Grained SPI (1) enforces *Stack Conservation* to defeat stack-backward pivoting to payloads within the active stack region (Figure 3.7(a)), and (2) sets the stale stack to zero in order to eliminate gadgets in stale region of the stack (Figure 3.7(b)), and therefore defeats stack-forward pivoting.

Note that SPI is significantly different from other stack protection schemes (e.g., [67]). Stack protection schemes are concerned with the integrity of the *contents* of the stack, whereas SPI enforces runtime invariants **P1, P2** in order to ensure the integrity of the stack pointer.

In the following, we first present the threat model that SPI addresses, then present an overview of coarse- and fine-grained SPI.

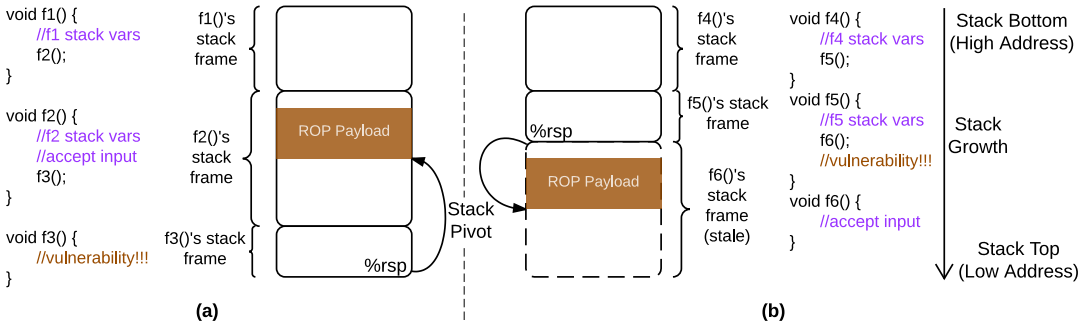

Fig. 3.7.: Fine grained SPI. (a) Stack-Backward Pivoting (b) Stack-Forward Pivoting

Threat Model Our solution assumes an adversary who has the capability to exploit a vulnerability in a program and achieve arbitrary code execution. Further, irrespective of ASLR, we assume that the adversary has full knowledge of the program layout and can

Table 3.3: SPI Defense Overview

SPI Granularity	**Type of Pivoting**	**Defense Technique**
Coarse	Outside stack segment	*StackPtr* bounds checking
Fine	Stack-Backward, As in Figure 3.7(a)	Shadow *StackPtr* stack
	Stack-Forward, As in Figure 3.7(b)	Zeroing of stale stack

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 [11] and [10] – that can successfully evade state-of-the-art binary-level CFI defenses.

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 SPI to be a fruitful defense against ROP is that the stack pivoting be required in order to carry out the attack. Our threat model is no more restrictive than state-of-the-art ROP defenses – if not lesser.

Coarse-Grarined SPI Coarse-grained SPI is an implementation of *Stack Localization* and addresses out-of-stack pivoting. Since stack pointer is indicative of the stack frame, **P1** is nothing but:

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

Pivoting outside the stack region requires the stack pointer to be altered by a large offset, which is suitably accomplished using an absolute SP-update instruction. Therefore,

in a nutshell, during coarse-grained SPI, we instrument each absolute SP-update instruction and assert that the new value of stack pointer lies within the stack region.

Fine-Grained SPI Fine-grained SPI is an implementation that defends against pivoting within the stack region. Each explicit SP-update instruction results in the movement of stack pointer. Under normal execution, movement of stack pointer along the direction of stack growth occurs during a function frame allocation and the reverse movement occurs during a function frame deallocation. *Stack Conservation* requires that the size of allocation is equal to the size of deallocation (**P2**). That is, for a given thread of execution:

$$StackPointer_{BeforeFunc()} == StackPointer_{AfterFunc()}$$

We maintain a per-thread shadow stack that keeps track of the stack pointer, and update the shadow stack after every explicit SP-update instruction. The value of stack pointer upon allocation is pushed on to the shadow stack. When a frame is deallocated, the value of stack pointer is required to match the value of stack pointer that was stored on the stack to ensure that there is no pivoting.

The shadow stack can only defend against attacks that violate *Stack Conservation*. That is, they employ stack-backward pivoting (Figure 3.7(a)), where the payload is located in the active region of the stack. In the case of stack-forward pivoting, one solution is to take an approach similar to the CFI-based defenses. That is, we could statically analyze the binary to generate a CFG and assert that the sizes of succeeding function frames are in accordance with the CFG. The problem with such a solution is that it inherits the coarse-grained nature of binary-level CFI approaches. Specifically the problems are two fold: first, it requires a complete CFG, therefore the solution is not incrementally

deployable. Second, due to lack of source code, one is forced to provide a coarse-grained policy, which, depending on the complexity of the program, is likely to provide sufficient attack space. That is, even with a conservative approach, we are likely to end up with several allowable frame sizes for succeeding functions in the CFG, thereby accommodating an attack space. We observe that the principle of *use-after-def* extends to function's stack frame:

Use-after-def (P3): *A function's stack frame can only be used after it is defined.* Stack-forward pivoting violates this requirement. **P3** must be true *irrespective* of the function that is invoked. We take a conservative approach and set the allocated stack region to 0, when a function's stack frame is allocated. This way, we effectively destroy any possible payload that may be present in the stale region of the stack. Details and optimizations are presented in Section 3.3.4.

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

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

The above JavaScript code loads hxds.dll, MS Office help library. hxds.dll 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. [21], but without any JIT code.

Elimination of Unintended SP-update Instructions As a final step, we eliminate all the unintended explicit SP-update instructions. SPI protects all the intended explicit SP-update instructions in the program. However, an attacker can utilize the unintended instructions that could result due to mis-aligned instruction access. Considerable research has gone into removing unintended gadgets from the program (e.g., G-Free [23], in-place code randomization [22]). We simply leverage these efforts to render the SPI-protected-binary free of explicit SP-updates. Unlike prior efforts, our threat model accommodates *any* sequence of instructions terminated by an indirect branch instruction as a potential gadget. Therefore, we eliminate unintended SP-update instructions as opposed to eliminating *all* the gadgets.

Furthermore, the number of unintended SP-update instructions are very few in number when compared to the number of gadgets considered for elimination by [22]. This is due to the vast number of one-byte indirect branch instructions (e.g., 0xc3 is a `ret` instruction) that gadget elimination approaches must eliminate. In comparison, most explicit SP-update instructions manifest as infrequent multi-byte instructions. Therefore, our modifications are less intrusive and often eliminate all the unintended explicit SP-write instructions.

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. For example, SPI – and other gadget elimination solutions – can not eliminate gadgets in such read-only data. However, a source code level implementation of SPI must ensure that no data is contained within executable regions.

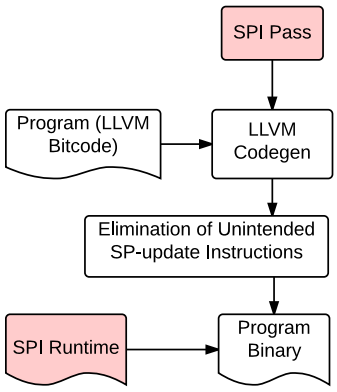

Fig. 3.8.: Work-flow of SPI

Work-flow of our defense is presented in Figure 3.8. The implementation comprises of SPI Pass, a LLVM code generation pass that performs instruction-level instrumentation to capture the explicit SP-updates, and a runtime that provides the implementation of the core functionality like assertion of **P1, P2**.

3.3.3 Coarse-Grained SPI

Algorithm The algorithm for coarse-grained SPI is presented in Algorithm 2. Given a *Program*, $EnforceCoarseSPI$ 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. The *StackPtr* is passed as an argument. Generally, displacement of stack pointer due to intended relative SP-update instructions is small when compared to the size of the stack region (8 MB on 64 bit Linux). Therefore, we limit the coarse-grained protection to absolute SP-update instructions. However, if tighter protection is needed, relative SP-update instructions can also be instrumented to invoke *CoarseCheck*.

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.

EnforceCoarseSPI is implemented within the SPI code generation pass, and the implementation dependent *CoarseCheck* is implemented within the target-dependent runtime.

Algorithm 2 Coarse-Grained SPI. Given the llvm bitcode *Program*, a call to *CoarseCheck* is inserted after each absolute SP-Update instruction.

```
1: procedure EnforceCoarseSPI(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
```

Coarse-grained SPI can protect against *all* attacks that contain non-stack payloads.

3.3.4 Fine-Grained SPI

Fine-Grained SPI defends against ROP attacks that contain stack-based payloads. Specifically, **P2** and **P3** are enforced to protect against stack-backward and stack-forward pivoting respectively.

Algorithm 3 presents the algorithm for fine-grained SPI. Similar to coarse-grained SPI, the instrumentation function: $EnforceFineSPI$ is implemented as a LLVM code generation pass (SPI Pass), and the target-dependent implementation of $GetSize$ and $UpdateSP$ are implemented within the runtime.

Stack-Pointer Shadowing We use a shadow stack to guard against stack-backward pivoting. Goal of the shadow stack is to track the movement of stack pointer across stack allocations and deallocations. Specifically, a thread-specific stack is created and stored within the TSD, and when a new stack frame is allocated, the value of stack pointer before allocation is pushed on to the shadow stack, and when the frame is deallocated, the value at the top of the shadow stack is popped, and the new value of stack pointer is checked against the popped value. A match ensures that stack frame is conserved.

In Algorithm 3, $UpdateSP$ determines if an SP-update instruction is an allocation or deallocation by comparing the current value of $StackPtr$ against previously encountered value. A lower value (assuming the growth of stack from higher to lower address) indicates allocation and a higher value indicates deallocation (line 20 in Algorithm 3).

In 64 bit Linux, the TSD structure can only be accessed through the `gs` segment register [1]. In order to prevent an attacker from gaining access to the TSD structure, during code generation and through elimination of unintended explicit SP-update instructions, we limit access to TSD within SPI runtimes.

In some cases, specifically during exception handling, it is possible that multiple stack frames are unwound and the execution does not resume at the immediate caller function. In such cases, the stack pointer is restored to an earlier stack frame using an absolute SP-update instruction. To accommodate such cases, if the value of the stack pointer during *UpdateSP* does not match the top of the stack, elements from the shadow stack are popped till a match is found. If no match is found, violation of **P2** is inferred.

Stale-Stack Zeroing We zero the stale (or inactive) stack region in order to defend against stack-forward pivoting. In principle, if a function frame utilizes x bytes on the stack, once the function returns, the x bytes are dead and ready for reuse. By zeroing the deallocated stack region, we ensure that any possible gadgets in the region are eliminated (**P3**). Zeroing the stack frames of *all* the functions upon return is redundant and performance intensive. Suppose all the functions in a binary: (1) contain only direct branch instructions, and (2) contain no absolute SP-update instructions, such a binary can not be used for stack pivot operation. Conversely, only functions that contain either indirect branch instruction or absolute SP-update instruction can be used for stack pivoting.

Pivot-Safe Functions: We define *Pivot Safe* functions, the functions that can not be used for stack-forward pivoting, and exclude such functions from zeroing. A function is *Pivot Safe* if:

1. It contains no absolute SP-update instructions, and

Algorithm 3 Fine-Grained SPI. Instrumentations are performed *after* the SP-Update instructions.

```
 1: procedure EnforceFineSPI(Program)
 2:     for each Function in Program do
 3:         if Function is not PivotSafe then
 4:             Save Live Registers
 5:             InsertCall zbytes ← GetSize(StackPtr)
 6:             InsertCodeToZero                ▷ Set zbytes bytes from StackPtr to 0
 7:             Restore Saved Registers
 8:         end if
 9:         for each Inst in Function do
10:             if Inst is SP-Update_Explicit then
11:                 Save Live Registers
12:                 InsertCall UpdateSP(StackPtr)
13:                 Restore Saved Registers
14:             end if
15:         end for
16:     end for
17: end procedure
18: procedure UpdateSP(StackPtr)
19:     Stack ← CreateOrGetShadowStackTLS()
20:     if StackPtr < Stack.Top then                ▷ Allocation
21:         Stack.Push(StackPtr)
22:         if StackPtr < Stack.Lowest then
23:             Stack.Lowest ← StackPtr
24:         end if
25:     else                                        ▷ Deallocation
26:         Stack.PopUntil(StackPtr)
27:         if Stack.Empty() then
28:             abort()                             ▷ Pivoting detected
29:         end if
30:     end if
31: end procedure
32: procedure GetSize(StackPtr)
33:     Stack ← GetShadowStackTLS()
34:     if Stack.Lowest < StackPtr then
35:         size ← (StackPtr − Stack.Lowest)
36:         Stack.Lowest ← StackPtr
37:         return size
38:     end if
39: end procedure
```

2. It contains no indirect branch instructions

Stack-forward pivot instructions are found in frame allocations. In pivot-safe functions, there is direct flow of control between each allocation and the corresponding deallocation of stack frame. Therefore, every potential stack-forward pivot instruction is met with an

opposite stack-backward pivot instruction that nullifies the effect of pivoting. All other functions are *Pivot Unsafe*.

It is not necessary to zero the stale stack before a pivot-safe function. When a pivot-unsafe function is encountered, it is possible that the allocated frame contains potential payload. It is necessary to zero the stale stack before the function executes. However, we are only required to zero the amount of stack that was used between the previous and current pivot-unsafe functions. In order to accomplish this, within the TSD structure, we also maintain the lowest value of the stack pointer (line 23 in Algorithm 3), which is the farthest the stack has grown between any two successive pivot-unsafe functions. During stack allocation within the prologue of a pivot-unsafe function, we first obtain the size of the stack that needs to be zeroed from *GetSize*, and then zero those many bytes from the stack pointer (line 6). *GetSize* accordingly adjusts the lowest point (line 36). This way, we vastly reduce the number of functions in which the stack must be zeroed. For example, in 64 bit gnu libc library, we found that over 70% of functions are pivot-safe.

An example of zeroing code (line 6): the one we use in our proof-of-concept implementation of SPI is as follows:

```
mov zbytes, %rax;
mov %rsp, %rdi;  //rsp is the StackPtr
mov %rax, %rcx;
shr $0x3, %rcx;
xor %rax, %rax;
rep stosq;
```

With precise analysis, it may be possible to exclude certain SP-unsafe functions as SP-safe (e.g., if it can be proven that the indirect branch instruction can not be used achieve stack-forward pivoting) thereby further reducing the number of functions that require stack zeroing. However, we do not conduct such analysis in this work.

Partial Zeroing: In theory, we do not have to zero all the stale stack region between 2 successive pivot unsafe functions. In fact, zeroing a "few" bytes starting from the stack pointer may be sufficient to render stack-forward pivoting useless. However, it is important to note that since implicit SP-update instructions are unprotected, an attacker could leverage intended or unintended sequence of implicit SP-update instructions *pop, retn* to walk-across the zeroed bytes and reach the payload. One solution would be to statically analyze and identify the maximum number of bytes an attacker could move the stack pointer using implicit SP-update instructions (intended or unintended), and set at least as many bytes to 0. This solution is a trade-off between performance and incremental-deploy ability because, it involves analysis of all the modules in the process memory, which may not be known ahead of time.

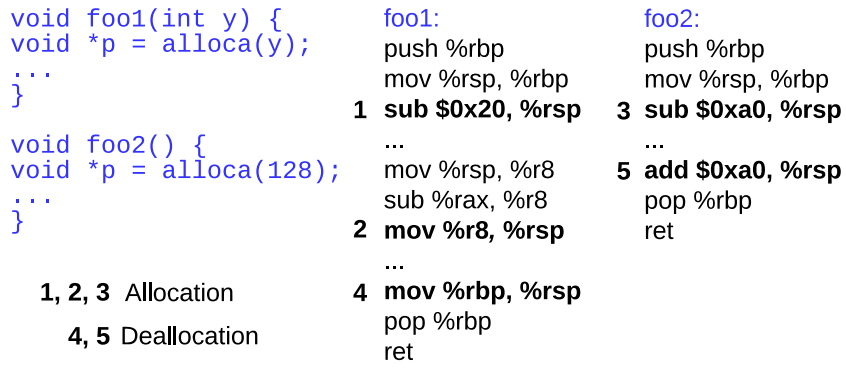

Fig. 3.9.: Dynamic allocation of stack space using `alloca`

Dynamically Allocated Stack Memory 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 3.9, 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 4), thereby satisfying *Stack Conservation* **(P2)**. However, in the case of `foo2`, the compiler statically determines the argument to `alloca` and allocates (and deallocates) 0x20 + 0x80 = 0xa0 bytes (markings 3 and 5) using relative SP-update instructions. In both cases, a call to $UpdateSP$ (as in Algorithm 3) is inserted after the explicit SP-update instructions in order to update the shadow stack correspondingly. When $UpdateSP$ is invoked after marking 4 in `foo1`, two elements are popped out of the shadow stack, one for each allocation at markings 1 and 2 respectively.

Explicit SP-update Injection through JIT Gadgets injected into Just-In Time (JIT) code by an attacker are particularly hard to protect against [21]. However, code generator

within a JIT engine can be modified to instrument all explicit SP-update instructions to enforce SPI. Also, all unintended SP-update instructions can be removed.

3.3.5 Discussion

Code-Reuse Attacks without Stack-Pivot 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, `pop` and `ret` instructions allow for movement of the stack pointer, which 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.

Schuster et al. introduce COOP [65], 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. However, because different virtual functions do not accept the same number of arguments, when executed from the same callsite, *Stack Conservation* property of SPI is violated, and can therefore be stopped by SPI.

3.4 Evaluation

In this section, we evaluate each of the three solutions presented in this dissertation with particular focus on attack space reduction. The performance evaluation and enforcement is presented in Chapter 4.

3.4.1 Challenges

Obtaining real-world exploits: Obtaining real-world exploits is a challenging task and not the focus of this dissertation. Therefore, in our work, we utilize synthetic exploit samples available through the Metasploit [68] penetration test framework. Note that this step does not affect the quality of evaluation since the vulnerabilities exploited by the samples from Metasploit are recent and real, and are used by real-world exploits.

Evaluating attack space reduction through SPI: SPI is a non-control-flow-based approach. Therefore, unlike semantic recovery based approaches, the idea of "allowable targets" for each indirect branch instruction does not hold. The primary means to evaluate SPI is the execution overhead (more in Chapter 4) it presents when compared to the lack of it. In this section, we evaluate the feasibility of SPI as a solution when compared against gadget elimination techniques.

3.4.2 Evaluation Test Set

We evaluate `Total-CFI` on a corpus of exploits from Metasploit. We include a kernel exploit to demonstrate the ability of `Total-CFI` to perform system-wide detection.

Furthermore, to evaluate `vfGuard`, we consider a set of C++ program modules presented in Tables 3.6, 3.7 and 3.8. Firstly, our test set comprises of programs containing between 100 and 2000 VTables, thereby providing sufficient complexity for analysis. Secondly, the modules in the test set are a part of popular browsers like Firefox and Internet Explorer, which are known to contain several vulnerabilities. SpiderMonkey is the

JavaScript engine employed by FireFox and Table 3.8 presents some of the modules used by Internet Explorer that contain reported vulnerabilities. Finally, the test set contains both open (Table 3.6, 3.7) and closed source programs (Table 3.8). While `vfGuard` operates on raw COTS binaries, open source programs provide the ground truth to evaluate `vfGuard`'s accuracy. Along with SpiderMonkey and the modules used by IE, the set consists of dplus browser, an open source browser and TortoiseSVN, an open source Apache subversion client for Windows.

Finally, to evaluate SPI, we consider popular opensourced suites such as GNU Binutils and Coreutils and SPEC-INT 2006 benchmark. We also evaluate the number of SP-update instructions in popular (and vulnerable) DDLs in Windows XP and Windows 7.

3.4.3 Recovering Function-Level Semantics

`Total-CFI` was implemented as a plugin for DECAF [69], which is a modification of Qemu [70] version 1.0.1, a full system emulator. Qemu offers transparency and the ability to monitor the entire system, and has been widely used [71, 72] in research. DECAF modifies dynamic translation code of Qemu to incorporate opcode specific callbacks into the translation blocks. It also modifies the TLB cache manipulation code to dispatch a callback whenever an entry is made to the TLB cache. In all, `Total-CFI` consists of 3.8K lines of C code. In this section, we present the evaluation of `Total-CFI`. All the experiments were performed on a system with Intel core i7, 2.93GHz Quad core processor and 8GB of RAM, running Ubuntu 10.04 with Linux kernel 2.6.32-44-generic-pae.

Name	Version	`.reloc` present?	*Fiber* present?	*Dyn Code* present?
Calculator	6.1	✗	✗	✗
Notepad	6.1	✗	✗	✗
Internet Explorer	8.0	✓	✓	✓
Firefox	3.5	✓	✗	✓
Adobe Reader	8.1.1	✗	✗	✓
Google Talk	1.0.72	✓	✗	✓
Microsoft Paint	6.1	✗	✗	✗
Windows Media Player	12.0	✓	✗	✓
XPS viewer	6.1	✓	✓	✓
Yahoo Messenger	8.1.0.29	✗	✗	✗
Apple Quicktime	7.69.80.9	✓	✗	✓
Apple iTunes	10.2	✓	✗	✗
Process Explorer	15.05	✓	✗	✗
Filezilla	0.9.40.0	✗	✗	✗
Google chrome	18.0.1025	✓	✗	✓
Windows Messenger	4.7	✗	✗	✗
RealPlayer	11	✗	✗	✓
DivX Player	6.2.5	✗	✗	✓
Winamp	5.2	✗	✗	✓
VLC Media Player	1.1.11	✗	✗	✓
Skype	5.10.0.116	✓	✗	✓
Registry Editor	6.1	✗	✗	✗

Table 3.4: False positives on Windows OS

CVE	Application (Version)	Attack Technique	Exploit EIP	Target EIP	Vulnerable Module
CVE-2010-0249	Internet Explorer (6.0)	Uninitialized memory. Heap spray	0x7dc98c85	0x0c0d0c0d	mshtml.dll
CVE-2010-3962	Internet Explorer (6.0)	Incorrect variable initialization. Heap spray	0x71a51440	0x71a52c66	mswsock.dll
CVE-2011-0073	FireFox 3.5.0	Dangling pointer abuse	0x00346e54	0x01730ee5	js3250.dll
CVE-2011-0257	QuickTime 7.6	Buffer overflow	0x0044888d	0x00194ab0	QuickTime-Player.exe
CVE-2006-1016	Internet Explorer (6.0)	Stack Overflow	0x773f67a8	0x0c112402	ws2_32.dll
CVE-2009-3672	Internet Explorer (6.0)	Incorrect variable initialization. Heap-spray	0x74913ff2	0x0013e0d4	mshtml.dll
CVE-2006-1359	Internet Explorer (6.0)	Incorrect variable initialization. Heap-spray	0x7c8097f3	0x77c3210d	mshtml.dll
CVE-2010-4398*	Windows 7 kernel	Improper driver interaction. Buffer overflow	0x95dca042	0xb8cb8694	win32k.sys

Table 3.5: Summary of Exploits

False-Positive evaluation To measure its accuracy, we tested benign applications and exploits on `Total-CFI` to check for false positives and false negatives. We ran `Total-CFI` on 25 common applications that are listed in Table 3.4, on Windows XP and Windows 7.

We observed that several pre-loaded application executables in Windows do not contain relocation table in them. For such executables, we parsed the PE file and extracted statically determinable addresses into the whitelist. 0 exploits were reported in all 25 applications.

False-Negative evaluation To check the effectiveness of `Total-CFI` on detecting exploits, we ran `Total-CFI` on 7 recent real-world exploits that have exploits available in the MetaSploit framework, and one Windows 7 kernel exploit. All the exploits were detected. The summary of exploits and their detection are listed in Table 3.5. It is worth noting that the kernel exploit, CVE-2010-4398, which starts as a user mode program, exploits a vulnerability in Win32k.sys and eventually escalates privilege. A crafted REG_BINARY value for `SystemDefaultEUDCFont` registry key is inserted to cause a stack-based buffer overflow in the *RtlQueryRegistryValues* function in Win32k.sys. Monitoring a user program (or a set of user programs) alone is insufficient in identifying such an attack. It is essential to monitor both the kernel code and the user level code to diagnose such attacks. Detailed results are tabulated in Table 3.5.

3.4.4 Recovering C++-Level Semantics

We implemented `vfGuard` in the following code modules. The CFI model generation part of `vfGuard` is implemented as a plugin for IDA-pro v6.2. An open source IDA-decompiler [61] was modified to perform data flow analysis for callsite identification. The platform consists of 5.6K lines of Python code and 3.4K lines were added to it.

We evaluated `vfGuard` in several respects. We first evaluated the accuracy of virtual callsite and VTable identification using several open source C++ programs, because source code is needed to obtain ground truth. Then, we measured the policy precision and compare with BinCFI [40] and SafeDispatch [4]. To evaluate the effectiveness of `vfGuard`, we tested multiple realworld exploits. Finally, we measured `vfGuard`'s coverage with respect to number of indirect branches protected, and performance overhead of policy enforcement.

Program	**Ground Truth**	`vfGuard`	**FP**	**FN**
SpiderMonkey	811	942	13.9%	0
dplus-browser_0.5b	270	334	19.1%	0
TortoiseProc.exe	568	595	4.7%	0

Table 3.6: VTable Identification accuracy.

Program	**Ground Truth**	`vfGuard`	**FP**	**FN**
SpiderMonkey	1780	1754	0	1.4%
dplus-browser_0.5b	309	287	0	7.1%

Table 3.7: Callsite Identification Accuracy

Identification Accuracy To ensure policy soundness, `vfGuard` must identify all legitimate VTables and must not identify any false virtual callsites. To measure the accuracy, we picked SpiderMonkey and dplus-browser for the Itanium ABI, and TortoiseProc for the MSVC ABI. We constructed the "ground truth" by using compiler options that dump the VTables and their layouts in the binary. Specifically, `-fdump-class-hierarchy` and `/d1reportAllClassLayout` compiler options were used to

compile the programs on g++ and Visual Studio 2013 respectively. The results are tabulated in Table 3.6. The compilers emit meta-data for (1) each class object's layout in the memory, and (2) each VTable's structure. We compared each of the VTables obtained from the ground truth against `vfGuard`. None of the legitimate VTables were missed in each of the cases. In all the cases, VTables identified by `vfGuard` contained some noise (from 4.7% to 19%). This was expected due to the conservative nature of `vfGuard`'s VTable scanning algorithm.

To evaluate callsite identification accuracy of `vfGuard`, we leveraged a recent g++ compiler option, `-fvtable-verify` [6] that embeds checks at all the virtual callsites in the binary to validate the VTable that is invoking the virtual call. We compiled SpiderMonkey with and without the checks, and matched each of the callsites that contained the compiler check to the callsites identified by `vfGuard`. Out of the functions that were successfully analyzed, `vfGuard` reported 0 false positives. It reported 1.4% and 7.1% false negatives (i.e., missed during identification) for SpiderMonkey and dplus-browser respectively.

These experiments indicate that the generated policies should be sound but a little imprecise, due to the noisy VTables and missing callsites.

Program	**Avg. Targets per CS (Basic Policy)**	**# Nested CS**	**Avg. Targets per CS (NCF)**	**Avg. Targets per CS (NCF+ CCF)**	**Estimated `call` Targets – BinCFI**	**Call Target Reduction w.r.t BinCFI**
ExplorerFrame.dll	231	257	227	223	8964	97.5%
msxml3.dll	96	219	88	84	6822	98.8%
jscript.dll	39	55	38	38	2314	98.4%
mshtml.dll	292	211	258	257	16287	98.3%
WMVCore.dll	268	562	256	244	8845	97.3%

Table 3.8: Average targets for the basic policy and the filters

Policy Precision To measure how precise our generated policies are, we generate policies for C++ binary modules in Internet Explorer 8. Table 3.8 presents the average number of targets per callsite under 3 configurations – basic policy, basic policy with Nested Callsite Filter (NCF) and basic policy with Nested Callsite Filter and Calling Convention Fiilter (CCF). Additionally, for each case, we estimated the number of targets in a policy generated by BinCFI. We included into the policy all the function entry points in the program. The exact reduction in the number of targets is tabulated in the last column. We can see that even with the basic policy generated by `vfGuard`, we were able to refine BinCFI's policy by over 95%. Here, the refinement numbers pertain to the virtual callsites protected by `vfGuard` and not all the indirect branch instructions within the module. An optimal defense will combine `vfGuard`'s policy for virtual callsites along with those generated by BinCFI (or CCFIR) for other branch instructions.

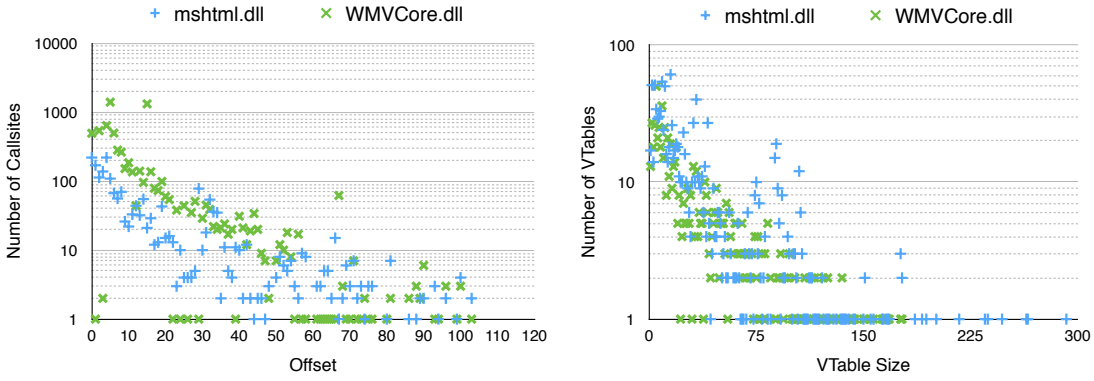

Fig. 3.10.: Distribution of callsites across various offsets

In general, we found no obvious correlation between the number of callsites and VTables in the binary to the effectiveness of the filters. While the filters improved precision in some cases, they did not in others. Graphs in Figure 3.10 show the scattered distribution of number of callsites with respect to offset within the VTable at the callsite,

and the number of VTables that contain a particular VTable size for `mshtml.dll` and `wmvcore.dll`. As expected, we found several VTables with small sizes of less than 10 elements. However, we also found a significant number of VTables between sizes 50 and 125. While the algorithm used by `vfGuard` is efficient in detecting the VTable start address, it is not very accurate in detecting the end points. This is the main hurdle for lowering the average call targets per callsite.

We also evaluate the precision of the policy generated by `vfGuard` as compared to SafeDispatch, a source code based solution. To estimate the policy produced by SafeDispatch, we modified g++ compiler option `-fvtable-verify` implementation. For every invoked callsite, it will insert code to output the number of possible targets. We compiled SpiderMonkey with this modified g++ compiler and ran its test cases. We observed that the average number of call targets per callsite is 109, and the maximum is 335. In comparison, `vfGuard` generated a policy for SpiderMonkey with 199 call targets per callsite on average and a maximum of 943 targets. This result indicates that the precision of CFI policies generated by `vfGuard` is within the same order of magnitude as those generated from a source code based solution.

CVE	**Target Application**	**Module**	**Remark**
2010-0249	Internet Explorer	mshtml.dll	Fake VTable
2013-1690	Firefox	xul.dll	Fake VTable
2011-1255	Internet Explorer	mshtml.dll	Fake VTable
2010-3962	Internet Explorer	mshtml.dll	Mis-aligned VTable access
2013-3893	Internet Explorer	mshtml.dll	Fake VTable

Table 3.9: Exploit mitigation. VTable based vulnerabilities.

Policy Effectiveness To assess the policy effectiveness, we conducted a survey on VTable related exploits. In particular, we found five working exploits towards Internet Explorer and Firefox, and listed them in Table 3.12. We tested the two exploits that target `mshtml.dll`. Being protected with the generated policy for `mshtml.dll`, Internet Explorer was able to detect these exploits successfully.

In CVE-2010-0249, an attacker sprays the heap with fake VTables and corrupts a stale object pointer by setting its vptr to the heap sprayed region. Then, at a callsite inside `mshtml.dll!CElement::GetDocPtr()`, attacker controlled vfptr is retrieved and executed. Since the attacker supplied vfptr is not a part of the policy for the callsite, it is flagged by `vfGuard` as an exploit.

In the case of CVE-2010-3962, `mshtml.dll` inadvertently increments the vptr of an object. So, in the virtual dispatch in `CLayout::EnsureDispNodeBackground()`, [address-point + `offset` + 1] is retrieved as the address of the vfptr instead of [address-point + `offset`]. Since the mis-aligned pointer does not belong to the policy, `vfGuard` flags it as an exploit. While we tested the above exploits successfully, more exploits in Table 3.12 follow the same modus operandi.

3.4.5 Quantifying Attack Space

Semantic recovery based approaches enforce CFI, which is dictated by the precision of the CFG. Attack space available to an attacker can be quantified by considering the possible number of targets at a particular indirect branch instruction.

Program	Avg. Targets per CS (DEP Only)	Avg. Targets per CS (Total-CFI)	Avg. Targets per CS (vfGuard)
ExplorerFrame.dll	842,091	8964	223
msxml3.dll	640,727	6,822	84
jscript.dll	548,490	2,314	38
mshtml.dll	4,585,953	16,287	257
WMVCore.dll	1,589,907	8,845	244

Table 3.10: Attack space reduction

Table 3.10 presents the reduction in attack space across two semantic recovery approaches presented in this thesis. Since `vfGuard` protects only the virtual callsites, we compare the attack space available between DEP-only, `Total-CFI` and `vfGuard` for virtual callsites. While `Total-CFI` was able to improve precision by eliminating 99.473% targets, `vfGuard` was able to eliminate 99.999% targets in comparison with DEP-only protection.

3.4.6 Stack-Pointer Integrity

We implemented a prototype for coarse- and fine-grained SPI. The instrumentation phase (Figure 3.8) 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). SPI LLVM pass comprises of 315 lines of C++ code for coarse-grained, and 2K lines of C++ code for fine-grained SPI. The runtime for coarse-grained and fine-grained SPI are 20 and 230 lines of assembly code, respectively.

Pivoting in Practice In Table 3.11, 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 3.12, we present a corpus of recent

Table 3.11: Absolute gadgets in Windows OS.

Program	**Gadget Module**	**Gadget Address**	**Pivot Instruction**	`.reloc?`	**SPI 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

exploits on Metasploits 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.

Table 3.12: 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`

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. SPI is particularly useful in protecting such non-relocatable modules.
Coarse-grained SPI can defeat pivoting in all cases listed in Table 3.11 except the gadget at uxtheme.dll:0x6ce8ab5e. uxtheme.dll contains read-only data interleaved with code in the `.text` segment, and data item `char s_keyPublic1[]` is at address 0x6ce8ab38. So SPI and other gadget-elimination solutions can not eliminate the gadget.

SPI vs Gadget Elimination In order to demonstrate the effectiveness of SPI when compared to gadget-elimination-based solutions, we list the number of explicit SP-update instructions that SPI needs to protect as opposed to the total number of gadgets in coreutils and binutils. The results are tabulated in Table 3.13. On one hand, we found that benign programs contain none or very few absolute SP-update instructions. On the other hand, the relative SP-update instructions are exclusive to function frame allocation and deallocation. Absolute SP-update instructions, the most popular for stack-pivoting are a very small fraction when compared to the total instructions in a program. Also, explicit SP-update instructions that SPI needs to protect are much smaller when compared to total number of gadgets that gadget elimination tools would need to eliminate.

3.5 Summary

In this section, we presented three integrity models in decreasing order of attack space.

Table 3.14 presents the scope of each of the integrity models proposed in this dissertation in comparison with shortcomings of current defenses as presented in Table 2.1.

Table 3.13: Explicit SP-Update instructions vs Gadgets

Suite	**Program**	**Total Instructions**	**Absolute SP-update Instructions**	**Relative SP-update Instructions**	**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 3.14: Scope of Integrity Models

Category	**Parameter**	`Total-CFI`	`vfGuard`	SPI
CFI	Precision	✓	✓	
	Deploy-ability			✓
	Runtime Performance		✓	✓
	Gadget Definition			✓
Artificial Diversity	Resilience			✓
	Diversification			✓
Gadget Elimination	Coverage			✓

A check mark (✓) indicates that the limitation is either eliminated or reduced by either improving the parameter or by circumventing it. For example, SPI is independent of diversification or definition of a gadget, therefore eliminates the limitation. Similarly, by improving the precision of the CFI model, `vfGuard` can reduce the runtime overhead of

enforcement because the number of indirect targets per virtual `call` instruction is reduced, thereby reducing the number of lookups.

4. INTEGRITY-MODEL ENFORCEMENT

Integrity model and its enforcement are independent of each other. That is, models can be independently enforced using one of many enforcement mechanisms. In this chapter, we present three different enforcement mechanisms we used in `Total-CFI`, `vfGuard` and SPI.

4.1 System-Wide Enforcement

We often do not know the vulnerable process or worse, processes that are exploited in a target system. In such cases, it is essential to monitor the entire system to record any violations that may occur. Moreover, system-wide enforcement has the advantage of monitoring the OS kernel, which has not received sufficient investigation. As a result, kernel exploit detection and diagnosis is still missing.

4.1.1 Performance Evaluation

`Total-CFI` was implemented as a plugin for DECAF [69], which is a modification of Qemu [70] version 1.0.1, a full system emulator. DECAF modifies dynamic translation code of Qemu to incorporate opcode specific callbacks into the translation blocks. It also modifies the TLB cache manipulation code to dispatch a callback whenever an entry is made to the TLB cache. In this section, we evaluate the performance of system-wide enforcement of `Total-CFI`. All the experiments were performed on a system with Intel core

Guest OS	Qemu	Total-CFI	Total-CFI + WL_Cache
WinXPSP3	48s	1m 27s	1m 15s
Win7SP1	1m 57s	3m 26s	3m 12s

Table 4.1: Times taken to boot Windows 7 and XP till the login screeen is reached

System state	# files in cache	Total Size (B)	Avg. size per file (KB)	# files without `.reloc`
Login screen	263	1725024	6.405	0
Desktop UI visible	385	2853392	7.237	7
Boot completed	454	3496120	7.52	9
3 programs running	504	3900312	7.557	9
5 programs running	645	5672224	8.588	13

Table 4.2: Memory Overhead for whitelist cache on Windows 7

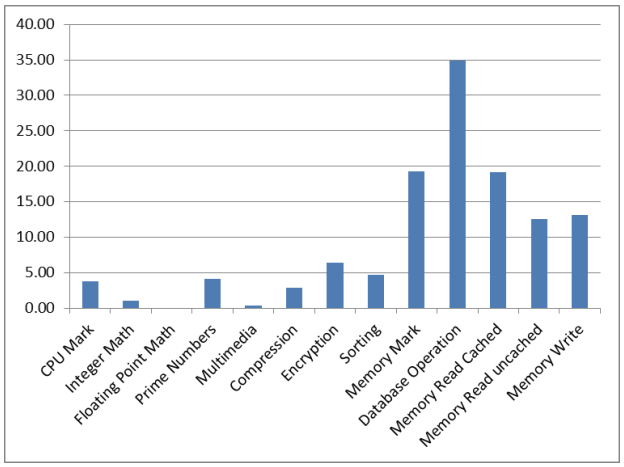

Fig. 4.1.: Performance of `Total-CFI` vs Qemu 1.0.1

i7, 2.93GHz Quad core processor and 8GB of RAM, running Ubuntu 10.04 with Linux kernel 2.6.32-44-generic-pae.

We capture the performance overhead introduced by `Total-CFI` under two categories. (1) Execution overhead and (2) Memory overhead. We conducted experiments to measure the boot time execution overhead introduced by `Total-CFI` on Windows 7 and XP. The

results are listed in Table 4.1. We consider boot time execution overhead under performance evaluation because variety of activities occur during the system boot that span across system level, user level and IO. Moreover, most module loads happen during the system boot. Therefore, the boot process is perhaps the worst case scenario with respect to performance overhead imposed by `Total-CFI`. Optionally, `Total-CFI` can be turned on after the boot process and it can monitor the execution of all the newly created processes from that point forward.

In Table 4.1 `Total-CFI` is configured in 2 modes, with and without caching of the whitelists. With whitelist caching enabled, the overhead on Windows XP and 7 were found to be 56.2% and 64.1% respectively. Keeping integration with hardware in mind, we also captured the memory overhead introduced by `Total-CFI` to maintain the shadow whitelist during the boot process on Windows 7. The memory overhead indicates the amount of memory required to store the whitelists. The results are tabulated in Table 4.2. We found the average overhead per file to be 7.46KB. We observed that the whitelist for files without `.reloc` section tend to be larger in size since `Total-CFI` takes a conservative approach to extract all the statically determinable addresses from the binary. Furthermore, from our experiments, we found that even with large number of programs in the memory (such as Microsoft Office applications, Adobe flash, IE, Google Chrome and so on), no more than 1000 files were present in the whitelist cache. At the rate of 7.46KB per file, one would need to set aside approximately 8MB in the hardware for the whitelists, which is conceivable. In combination with a carefully designed cache flush policy to accommodate for even larger number of files in the memory, we believe that integrating whitelist management into the hardware is not far fetched.

Furthermore, we ran the Pass Mark [73] CPU and Memory benchmark on `Total-CFI`. The results are shown in Figure 4.1. The CPU benchmark on `Total-CFI` showed an average overhead of 4.4% over Qemu and the memory benchmark showed an average overhead of 19.8%.

4.2 Process-Level Enforcement

In process-level enforcement, the monitoring is restricted to a particular process. Because the kernel is not monitored, performance of process-level monitoring is better than whole-system monitoring.

We enforced the policy generated by `vfGuard` by running the program on Pin [74]: a dynamic binary translator. A PinTool was written using 850 lines of C++ code to perform policy enforcement. In our proof-of-concept approach, we intercept the control flow at every previously identified callsite and check if the call target is allowable for the callsite. If the target is dis-allowed, the instance is recorded as a violation of policy. Effective enforcement must impose low space and runtime overheads. Under the basic policy, `vfGuard` captures the policy within 2 maps. The first map M_{cs} maps a callsite to the VTable offset at the callsite, and the second map M_{target} maps a given target to a 160-bitvector[1] that represents the valid VTable offsets for the given target. That is, i^{th} bit set to 1 indicates that the target is valid for offset $i * 4$. For a given CS, a M_{cs} entry is readily derived from τ_{val} in Equation 3.1 in Section 3.2.4. M_{target} is populated from the identified VTables (Section 3.2.5). For each VTable entry, the corresponding bitvector is

[1]The size of the bitvector is dictated by the size of the largest VTable in the binary. We found 160 to be sufficient.

updated to indicate a 1 for the offset at which the entry exists within the VTable. `vfGuard` performs 2 map lookups and 1 bitvector masking to verify the legitimacy of a given target at a callsite. That is, for a given callsite (CS) and target (T), the target is validated if:

$$BitMask(M_{cs}(CS)) \ \& \ M_{target}(T) \neq 0 \tag{4.1}$$

Such a design enables quick lookup and limits space overhead from duplication of callsites and targets within the maps. While the map lookups and bitvector masking result in constant time runtime overhead, the space requirements of M_{cs} and M_{target} are linear with resepct to number of callsites and targets respectively.

In case of callsites whose targets were filtered, the target lookup is different from basic policy. Each callsite CS – whose targets were filtered – is associated with a map $M_{Filtered}(CS)$ that maintains all the allowable call targets for CS. During enforcement, `vfGuard` first checks if the callsite is present in $M_{Filtered}(CS)$ and validates the target. If callsite is not present in $M_{Filtered}$ (i.e., targets for the callsite were not filtered), `vfGuard` performs the 2 map lookup and verifies the target through Equation 4.1. Enforcing the filtered policy introduces greater space overhead. The main reason being: targets reappear in multiple $M_{Filtered}$ for each of the callsites that the targets are valid at. We wish to investigate better enforcement in our future work.

Effect of Module-Level ASLR: `vfGuard` performs policy enforcement with or without module-level ASLR enabled. At a module granularity, the base addresses of the modules are randomized. That is, the modules are loaded at different addresses during each

instance of loading. The callsite and targets in M_{cs}, M_{target} and $M_{Filtered}$ are stored as (module, offset) tuple rather than the concrete virtual address. When a module is loaded, the virtual addresses of callsite and target addresses are computed from the load address of the module.

4.2.1 Cross-Module Inheritance

In practice, classes in one module can inherit from classes defined in another module [75]. Therefore, as new modules are loaded into a process address space, the allowable call targets for callsites in existing modules need to evolve to accommodate the potential targets in the new module. Given a list of approved modules that a program depends on, `vfGuard` can analyze each of the modules to generate the intra-module policy. From the execution monitor, `vfGuard` monitors module loads to capture any newly loaded modules and their load addresses. Intra-module policies are progressively adjusted (for ASLR) and maps M_{cs}, M_{target} and $M_{Filtered}$ are updated so as to capture the allowable targets for various callsite offsets across all approved modules. If a target in an unapproved modules is invoked, `vfGuard` records it as a violation.

4.2.2 Performance Evaluation

`vfGuard` performs policy enforcement using PinTool, a publicly available process-level runtime execution monitor. To measure the performance overhead imposed by `vfGuard`, we opened load-intensive webpages on Internet Explorer and recorded the overhead on 3 individual modules with respect to Pin as baseline. The results are graphed in Figure 4.2.

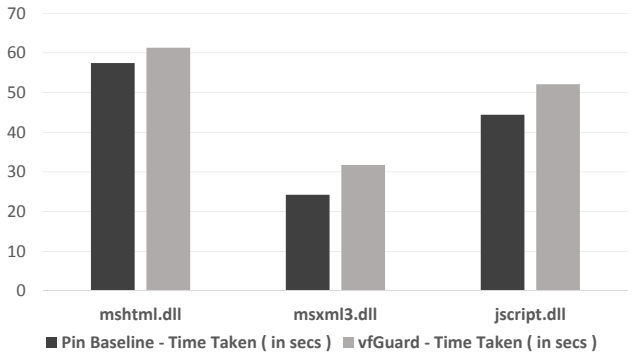

Fig. 4.2.: Performance overhead imposed by `vfGuard`

Overall, we found an average overhead of 18.3%. We made no attempt to optimize the runtime performance of policy enforcement. While this performance overhead is not impressively low, it is aligned with other binary-level CFI protection solutions, such as BinCFI.

4.3 IR-Level Compile-Time Enforcement

In SPI, the enforcement is performed at the LLVM IR level. IR-level enforcement is accomplished purely through static analysis, and therefore has the advantage of not requiring any changes to the system. Moreover, the LLVM IR can either be derived from the program source code using a front-end like clang, or by lifting the binary using a platform like McSema [76].

4.3.1 Performance Evaluation

We evaluate the performance of coarse- and fine-grained SPI on SPEC-INT 2006 benchmark, and performance of coarse-grained SPI on GNU coreutils (ver 8.23.137) and GNU binutils (ver 2.25). The results for SPEC benchmark are presented in Figure 4.3 and 4.4, and results for coreutils and binutils are presented in Figure 4.5 and 4.6. Overall, we found that coarse-grained SPI imposes very little overhead. Average overhead of coarse-grained SPI 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.

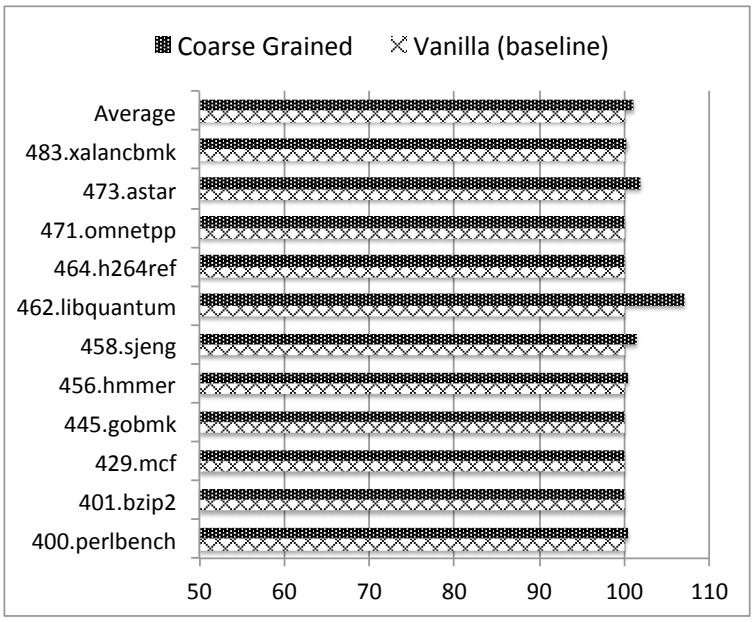

Fig. 4.3.: SPEC INT 2006 performance benchmark for coarse-grained SPI

As expected the overhead imposed by fine-grained SPI was higher, with an average of 29.93%. We found two main causes for the overhead. Firstly, C++ programs tend to

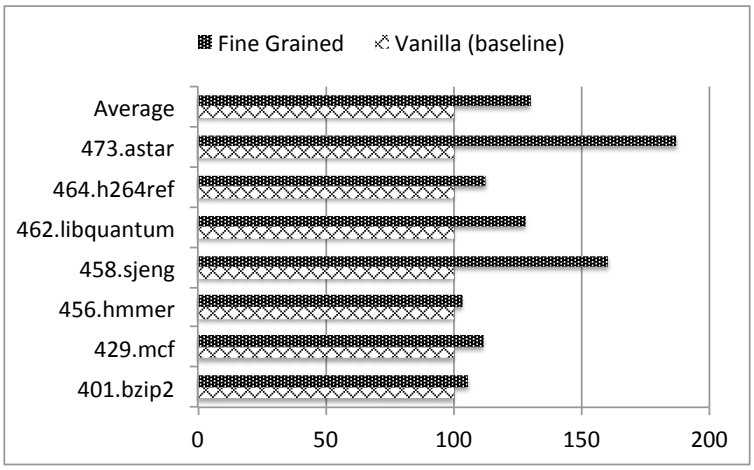

Fig. 4.4.: SPEC INT 2006 performance benchmark for fine-grained SPI

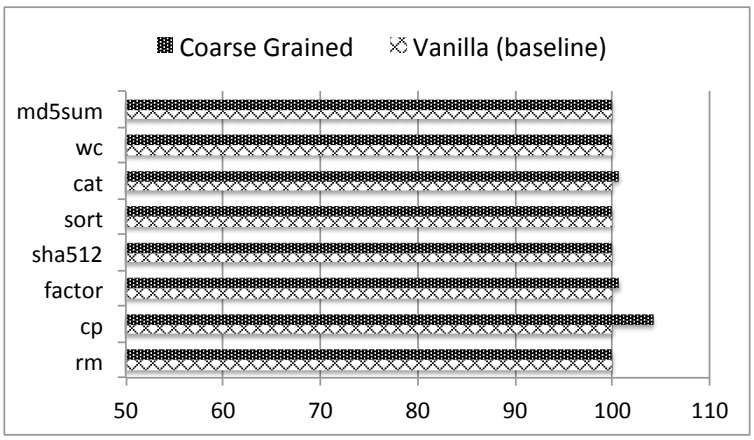

Fig. 4.5.: Coarse-grained SPI performance for GNU Coreutils

contain several virtual function dispatches, which are implemented using indirect `call` instruction. Each function that invokes a virtual function is pivot-unsafe, and therefore results in frequent zeroing. For example, in Figure 4.4, astar, which is a C++ program imposes the most overhead of 87.16%. Secondly, a significant part of the overhead occurs due to zeroing, which is $O(n)$ with respect to the amount of space to be set to zero. In astar, 52% of the overhead occurs due to zeroing. At the cost of deploy-ability, the cost of

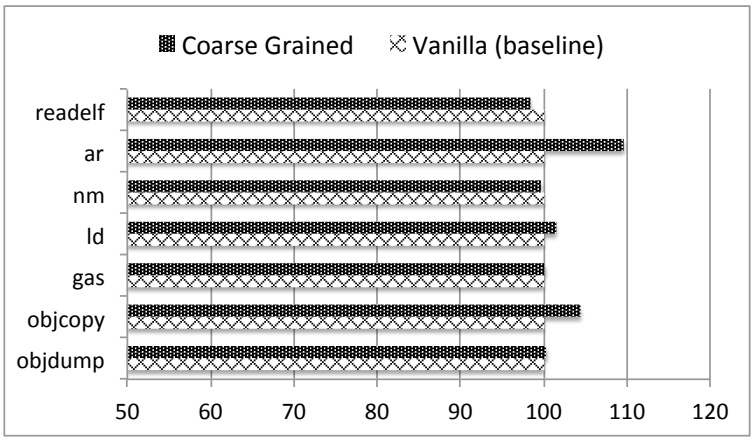

Fig. 4.6.: Coarse-grained SPI performance for GNU Binutils

zeroing can be lowered by incorporating the "partial zeroing" technique described in Section 3.3.4.

4.4 Other Enforcement Strategies

Other enforcement techniques such as Binary Rewriting([77] and [78]) as used by [24, 30, 40], in-memory enforcement by injecting code into process memory (e.g., using Browser Helper Objects [79]), etc. are equally feasible. Prior approaches (e.g., [24, 30, 40]) have leveraged static instrumentation to introduce Inline Reference Monitors (IRMs) to check the legitimacy of a branch target at runtime. We believe such approaches can improve the performance of `vfGuard`. Furthermore, depending on the number of modules loaded, the size of callsite and target maps can increase to result in significant memory overhead, specially in case of filtered targets. In such cases, cross-module dependencies can be analyzed to only allow cross-module calls in cases where known dependencies exist, thereby controlling the size of various enforcement maps.

5. LIMITATIONS AND FUTURE WORK

In this section we discuss the limitations of each of the solutions described in Chapters 3 and 4, and derive motivation for future work.

5.1 Attack Space in `vfGuard`

While `vfGuard` improves the precision with respect to state-of-the-art binary-level defenses like BinCFI, it still has several inaccurate edges in the CFG. For example, in Figure 3.5, while ground truth contains 6 function pointers, `vfGuard` identifies 3 VTables with a total of 6+4+2 = 12 function pointers. This results in an attack space that only increases with the number of VTables in the system. In future, we intend to investigate techniques to improve VTable demarcation.

COOP [65] demonstrates an attack that leverages such redundancy in VTable-based defenses. However, it is important to note that the nature of defense provided by `vfGuard` ensures that turing-completeness can not be achieved within the attack space [65].

5.2 Low precision due to indirect `jmp` and `ret` instructions

`vfGuard` only improves the precision for indirect `call` instructions. From Table 5.1, we can see that indirect `call` instructions form a significant fraction of the overall indirect branch instructions. However, other indirect branch instructions – indirect `jmp` and `ret`

Program	# Indirect `call` instructions	# Indirect `jmp` instructions	# `ret` instructions	Total # Indirect branch instructions
ExplorerFrame.dll	7797 (51.2%)	87	7266	15227
msxml3.dll	5439 (46.6%)	78	6157	11674
jscript.dll	2235 (33.5%)	5	4430	6670
mshtml.dll	9843 (38.3%)	352	15479	25674
WMVCore.dll	9748 (53.3%)	50	8497	18295

Table 5.1: Profile of indirect branch instructions

instructions continue to suffer from low precision. As a part of future work, we will investigate generic approaches that recover high-level semantics to impose restrictions on *all* indirect branch instructions, and not just indirect call instructions.

5.3 Stack-Pointer-Aligned Payload

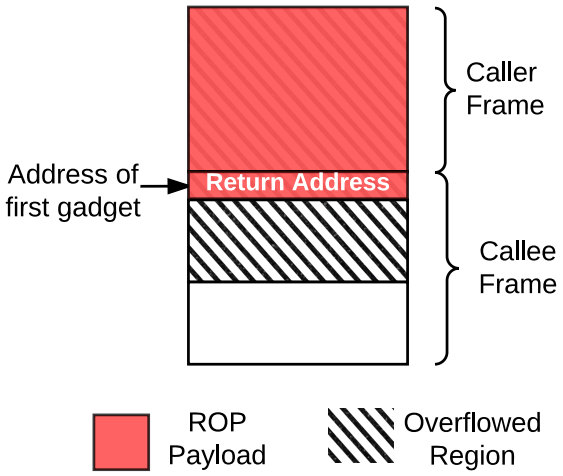

Fig. 5.1.: Stack-pointer-aligned payload

SPI 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 attack). For example, in Figure 5.1, 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, [5, 80, 81]) with practical implementations. StackGuard [80], 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.

5.4 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. In fact, preventing pivoting through implicit SP-update instructions presents two hard challenges. Firstly, several unintended implicit

SP-update instructions like `pop` *`reg`*, `push` *`reg`*, `ret`, etc., which are all one-byte instructions can be found in the memory. Eliminating each of them would compound the complexities faced by gadget elimination approaches [22, 23]. Secondly, tracking the movement of stack pointer at each of the implicit instructions would introduce a much higher overhead than fine-grained SPI. This is the reason we limit SPI to tracking explicit SP-update instructions.

However, as shown in Figure 1.1, SPI, in combination with CFI approaches can address the problem of pivoting through implicit SP-update instructions.

5.5 Future Work

The redundancy in the CFI model generated by `vfGuard` arises due to two reasons: (1) Incorrect VTable bounds, and (2) Lack of precise hierarchy information. In future, we would like to perform analysis to more precisely generate class hierarchy information. Particularly, corresponding entries in any two polymorphic VTables must be polymorphic to each other. Therefore, the *type* of corresponding arguments must be compatible with each other. We plan to leverage type analysis to generate more precise class hierarchy.

Furthermore, by combining training and theorem proving, we wish to recover more semantics, specially the scope (i.e., visibility) related semantics like public, private, protected, etc. in C++ binaries and incorporate them to generate a more precise CFI model.

6. CONCLUSION

In this dissertation, we sought to reduce the attack space in code-reuse attacks through program integrity models. Particularly, we operated directly on the binary and addressed several challenges including recovery of high-level semantics.

We proposed `Total-CFI`, which recovered function-level semantics from the binary to generate more precise CFI model. Further, we proposed `vfGuard`, which recovers C++-level semantics to provide a more precise CFI model for C++ binaries. In particular, it protects C++ virtual function calls in the binary. In an attempt to further reduce the attack space, we observed that stack pointer assumes the role of the instruction pointer in code-reuse attacks. Therefore, CFI in regular execution domain is analogous to SPI in code-reuse attacks. We implemented the SPI model, which provides strict protection with low overhead.

Finally, we presented three modes of enforcement of the integrity models. `Total-CFI` performed *system-wide* enforcement that enabled *whole-system* protection. `vfGuard` performed process emulation by enforcing at a process level, and finally, SPI incorporated the security checks by performing alterations the the LLVM-IR level.

LIST OF REFERENCES

[1] "Data execution prevention." http://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx.

[2] N. News, "Sony hack." http://www.nbcnews.com/storyline/sony-hack, 2014.

[3] E. C. Nicolas Falliere, Liam O Murchu, "Symantec stuxnet dossier." `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[4] 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.

[5] 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*, pp. 275–288, 2002.

[6] 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)*, pp. 941–955, 2014.

[7] 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*, pp. 552–561, ACM, 2007.

[8] J. Salwan, "Ropgadget tool, 2012," *URL http://shell-storm. org/project/ROPgadget*.

[9] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy.," in *20th USENIX Security Symposium*, 2011.

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

[11] 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.

[12] 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*, vol. 3, pp. 105–120, 2003.

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

[14] S. Bhatkar and R. Sekar, "Data space randomization," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 1–22, Springer, 2008.

[15] H. Xu and S. J. Chapin, "Address-space layout randomization using code islands," *Journal of Computer Security*, vol. 17, no. 3, pp. 331–362, 2009.

[16] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Tech. Rep. CMU-CS-02-197, Carnegie Mellon University, 2002.

[17] 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)*, pp. 157–168, ACM, 2012.

[18] 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)*, 2015.

[19] H. Xu and S. J. Chapin, "Improving address space randomization with a dynamic offset randomization technique," in *Proceedings of the ACM symposium on Applied computing*, pp. 384–391, ACM, 2006.

[20] 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*, pp. 298–307, ACM, 2004.

[21] 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 (S&P'13)*, pp. 574–588, 2013.

[22] 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 (S&P'12)*, pp. 601–615, IEEE, 2012.

[23] 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 (ACSAC'10)*, pp. 49–58, ACM, 2010.

[24] 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)*, pp. 340–353, 2005.

[25] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[26] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[27] 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.

[28] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of 21st ACM Conference on Computer and Communication Security (CCS '14)*, 2014.

[29] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in *Proceedings of the 15th Annual USENIX Security Symposium (Usenix Security'06)*, 2006.

[30] 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)*, pp. 559–573, 2013.

[31] 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.

[32] 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.

[33] K.-s. Lhee and S. J. Chapin, "Type-assisted dynamic buffer overflow detection.," in *USENIX Security Symposium*, pp. 81–88, 2002.

[34] 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.

[35] H. Xu, W. Du, and S. J. Chapin, "Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths," in *Recent Advances in Intrusion Detection*, pp. 21–38, Springer, 2004.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing.," in *USENIX Security*, 2013.

[37] 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.

[38] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proceedings of the 23rd USENIX conference on Security Symposium*, 2014.

[39] M. Zhang, A. Prakash, X. Li, Z. Liang, and H. Yin, "Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis," in *Proceedings of 19th Annual Network & Distributed System Security Symposium*, 2012.

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

[41] "Address space layout randomization (aslr)." http://pax.grsecurity.net/docs/aslr.txt, 2003.

[42] 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)*, pp. 311–322, 2013.

[43] M. Pietrek, "Msdn magazine, march 2002: An in-depth look into the win32 portable executable file format, part 2." `http://msdn.microsoft.com/en-us/magazine/cc301808.aspx`.

[44] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals. 5th Ed.* Microsoft press, 2009.

[45] 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 ACM conference on Computer and communications security*, CCS '12.

[46] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, 2005.

[47] A. Chaudhuri, P. Naldurg, and S. Rajamani, "A type system for data-flow integrity on windows vista," *SIGPLAN Not.*, vol. 43, pp. 9–20, Feb. 2009.

[48] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*.

[49] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, USENIX Association, 2006.

[50] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation.," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.

[51] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS'07)*.

[52] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*.

[53] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "On the trustworthiness of memory analysis – an empirical study from the perspective of binary execution," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2014.

[54] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pp. 1–12, June 2013.

[55] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: subverting virtual machine introspection for fun and profit," in *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS'10)*, 2010.

[56] 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.

[57] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 4th ed., 2013.

[58] "Itanium C++ ABI." http://mentorembedded.github.io/cxx-abi/abi.html.

[59] J. Ray, "C++: Under the hood." http://www.openrce.org/articles/files/jangrayhood.pdf, March 1994.

[60] "Stack Shield." http://www.angelfire.com/sk/stackshield/.

[61] F. Chagnon, "IDA-Decompiler." https://github.com/EiNSTeiN-/ida-decompiler.

[62] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[63] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code.," in *Proceedings of 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, 2012.

[64] Nektra, "Vtbl – IDA plugin." https://github.com/nektra/vtbl-ida-pro-plugin, 2013.

[65] 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.

[66] 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*, pp. 30–40, ACM, 2011.

[67] 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)*.

[68] "Metasploit penetration testing framework." `http://www.metasploit.com`.

[69] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 248–258, ACM, 2014.

[70] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[71] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin, "Os-sommelier: memory-only operating system fingerprinting in the cloud," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12.

[72] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin, "Multi-aspect, robust, and memory exclusive guest os fingerprinting," *IEEE Transactions on Cloud Computing*, 2014.

[73] "Passmark benchmark." http://www.passmark.com, 2014.

[74] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pp. 190–200, 2005.

[75] "Using dllimport and dllexport in C++ classes." http://msdn.microsoft.com/en-us/library/81h27t8c.aspx.

[76] A. Dinaburg and A. Ruef, "McSema: Static Translation of X86 Instructions to LLVM." http://recon.cx/2014/slides/McSema.pdf, 2014.

[77] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a distributed environment," Tech. Rep. MSR-TR-2001-50, Microsoft Research, April 2001.

[78] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for linux," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*, March 2010.

[79] "Browser Helper Objects." http://sysinfo.org/bhoinfo.html.

[80] 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*, vol. 98, pp. 63–78, 1998.

[81] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities.," in *Symposium on Network and Distributed System Security (NDSS'00)*, pp. 2000–02, 2000.

VITA

Aravind Prakash is a PhD student in the Department of Electrical and Computer Engineering at Syracuse University. His research interests lie in system security with emphasis on binary analysis.

In 2004 Aravind obtained a degree of Bachelor of Engineering from Visvesvaraya Technological University, Belgaum, India, and in 2009, he obtained a degree of Master of Science from University of Miami, FL, USA. After obtaining his PhD, Aravind will join the Department of Computer Science at SUNY Binghamton as an assistant professor.