Control-Flow Integrity

Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily controlling program behavior. CFI enforcement is simple, and its guarantees can be established formally, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protected shadow call stack and of access control for memory regions.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.2.4 [**Software Engineering**]: Software/Program Verification; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Languages, Verification

Keywords

Binary Rewriting, Control-Flow Graph, Inlined Reference Monitors, Vulnerabilities

1. INTRODUCTION

Computers are often subject to external attacks that aim to control software behavior. Typically, such attacks arrive as data over a regular communication channel and, once resident in program memory, they trigger a pre-existing software flaw. By exploiting such flaws, the attacks can subvert execution and gain control over software behavior. For instance, a buffer overflow in an application may result in a call to a sensitive system function, possibly a function that the application was never intended to use. The combined effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many ingenious vulnerability mitigations have been proposed for defending against these attacks; these include stack canaries [14], runtime elimination of buffer overflows [46], randomization and artificial heterogeneity [41, 62], and tainting of suspect data [55]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attackers have found ways to circumvent each deployed mitigation mechanism [42, 49, 61].

The limitations of these mechanisms stem, in part, from the lack of a realistic attack model and the reliance on informal reasoning and hidden assumptions. In order to be trustworthy, mitigation techniques should—given the ingenuity of would-be attackers and the wealth of current and undiscovered software vulnerabilities—be simple to comprehend and to enforce, yet provide strong guarantees against powerful adversaries. On the other hand, in order to be deployable in practice, mitigation techniques should be applicable to existing code (preferably even to legacy binaries) and incur low overhead.

This paper describes and studies one mitigation technique, the enforcement of *Control-Flow Integrity* (CFI), that aims to meet these standards for trustworthiness and deployability. The paper introduces CFI enforcement, presents an implementation for Windows on the x86 architecture, gives results from experiments, and suggests applications.

The CFI security policy dictates that software execution must follow a path of a *Control-Flow Graph* (CFG) determined ahead of time. The CFG in question can be defined by analysis—source-code analysis, binary analysis, or execution profiling. For our experiments, we focus on CFGs that are derived by a static binary analysis. CFGs can also be defined by explicit security policies, for example written as security automata [17].

A security policy is of limited value without an attack model. In our design, CFI enforcement provides protection even against powerful adversaries that have full control over the entire data memory of the executing program. This model of adversaries may seem rather pessimistic. On the other hand, it has a number of virtues. First, it is clear, and amenable to formal definition and analysis. It also allows for the real possibility that buffer overflows or other vulnerabilities (e.g., [26]) would lead to arbitrary changes in data memory. Finally, it applies even when an attacker is in active control of a module or thread within the same address space as the program being protected.

Whereas CFI enforcement can potentially be done in several ways, we rely on a combination of lightweight static verification

CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.

and machine-code rewriting that instruments software with runtime checks. The runtime checks dynamically ensure that control flow remains within a given CFG. As we demonstrate, machine-code rewriting results in a practical implementation of CFI enforcement. This implementation applies to existing user-level programs on commodity systems, and yields efficient code even on irregular architectures with variable-length instruction encodings. Although machine-code rewriting can be rather elaborate, it is simple to verify the proper use of instrumentation in order to ensure inlined CFI enforcement.

CFI enforcement is effective against a wide range of common attacks, since abnormal control-flow modification is an essential step in many exploits—independently of whether buffer overflows and other vulnerabilities are being exploited [42, 61]. We have examined many concrete attacks and found that CFI enforcement prevents most of them. These include both classic, stack-based buffer-overflow attacks and newer, heap-based "jump-to-`libc`" attacks. They also include recently described "pointer subterfuge" attacks, which foil many previous mitigation techniques. Of course, CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG (e.g., [10]) are not prevented. These include, for example, certain exploits that rely on incorrect argument-string parsing to cause the improper launch of a dangerous executable.

No matter how the CFG is defined or how permissive it is, CFI can be used as a foundation for the enforcement of more sophisticated security policies, including those that prevent higher-level attacks. For example, CFI can prevent the circumvention of two well-known enforcement mechanisms, *Inlined Reference Monitors* (IRMs) and *Software Fault Isolation* (SFI) [16, 17, 60]. In particular, CFI can help protect security-relevant information such as a shadow call stack [22, 37, 43], which can be used for placing tighter restrictions on control flow. Further, CFI can serve as the basis of a generalized, efficient variant of SFI that we call *Software Memory Access Control* (SMAC), which is embodied in an inlined reference monitor for access to memory regions. SMAC, in turn, can serve for eliminating some CFI assumptions.

Concretely, we develop fast, scalable implementations of CFI. We focus on one that provides strong guarantees and applies to existing x86 Windows binaries. Its performance on popular programs, including the SPEC benchmark suite, gives evidence of its efficiency. Building on CFI, we develop an implementation of a protected user-level shadow call stack. To the best of our knowledge, this implementation is an order-of-magnitude faster than previous software implementations with the same level of protection. Since SFI for x86 has been relatively slow and complex, we also examine the overhead of a simple CFI-based method for enforcing the standard SFI policy on x86; again, our measurements indicate an order-of-magnitude overhead reduction.

We have formally proven the correctness of inlined CFI enforcement for an abstract machine with a simplified instruction set. This formal treatment of inlined CFI enforcement contributes to assurance and served as a guide in our design. We have also analyzed a combination of CFI and SMAC, similarly.

The next section, Section 2, discusses related work. Section 3 informally explains CFI and its inlined enforcement. Section 4 describes our main CFI implementation and gives performance results. It also reports on our security-related experiments. Section 5 shows how additional security enforcement can be built on CFI; it includes a discussion of IRMs and three important examples: faster SFI, SMAC, and a protected shadow call stack. Section 6 summarizes our formal work; see [1] for further details. Finally, Section 7 concludes.

2. RELATED WORK

Our work on CFI is related to many techniques that, either directly or indirectly, constrain control flow. For the purposes of the present section, we divide those techniques according to whether they aim to achieve security or fault-tolerance.

2.1 CFI and Security

Constraining control flow for security purposes is not new. For example, computer hardware has long been able to prevent execution of data memory, and the latest x86 processors support this feature. At the software level, several existing mitigation techniques constrain control flow in some way, for example by checking stack integrity and validating function returns [14, 43], by encrypting function pointers [13, 62], or even by interpreting software using the techniques of dynamic machine-code translation [29].

Clearly, this a crowded, important research area (e.g., [5, 7, 9, 12, 15, 22, 30, 33, 37, 39, 41, 46, 55, 56]). Next we elaborate on some of the pieces of work most closely related to ours. In short, we believe that the distinguishing features of CFI are its simplicity, its trustworthiness and amenability to formal analysis, its strong guarantees even in the presence of a powerful adversary with full control over data memory, and its deployability, efficiency, and scalability. Like many language-based security techniques, but unlike certain systems for intrusion detection, CFI enforcement cannot be subverted or circumvented even though it applies to the inner workings of user-level programs (not just at the system call boundary).

SFI and Inlined Reference Monitors. IRMs are a general technique for enforcing fine-grained security policies through inlined checks [16, 17]. SFI is an important, special IRM that performs dynamic checks for the purposes of memory protection [17, 34, 50, 60]. SFI and other IRMs operate by adding code for security checks into the programs whose behavior is the subject of security enforcement.

IRM implementations must consider that a subject program may attempt to circumvent the added checks—for example, by jumping around them. As a result, IRM implementations typically impose restrictions on control flow. The necessary restrictions are weaker than CFI.

Those difficulties are compounded on hardware architectures that use variable-length sequences of opcode bytes for encoding machine-code instructions. For example, on x86 Linux, the machine code for a system call is encoded using a two-byte opcode sequence, `CD 80`, in hexadecimal, while the five-byte opcode sequence `25 CD 80 00 00` corresponds to the arithmetic operation `and eax, 80CDh`. Therefore, on x86 Linux, if this particular `and` instruction is present in a program, then jumping to its second opcode byte is one way of performing a system call. Similarly, other x86 instructions, such as those that read or write memory, may be executed through jumps into the middle of opcode byte sequences.

As a result, existing implementations of IRMs for the x86 architecture restrict control flow so that it can go only to the start of valid instructions of the subject programs. In particular, control flow to the middle of checking sequences, or directly to the instructions that those sequences protect, is prohibited.

The performance of IRMs has been problematic, in large measure because of the need for control-flow checks, particularly on the x86 architecture [17, 34, 50]. CFI offers an alternative, attractive implementation strategy; its guarantees, while stronger than strictly necessary for IRMs, imply the required control-flow properties, and thereby CFI can serve as a foundation for efficient IRM implementation. We elaborate on some of these points in Section 5.

Vulnerability Mitigation Techniques with Secrets. PointGuard [13] stores code addresses in an encrypted form in data memory. The intent is that, even if attackers can change data memory, they cannot ensure that control flows to a code address of their choice: for this, they would have to know the corresponding decryption key. Several other techniques [7, 14, 41, 56, 62] also rely on secret values that influence the semantics of pointer addresses stored in memory, For instance, PaX ASLR shuffles the virtual-address-space layout of a process at the time of its creation, using a random permutation derived from a per-process secret [41]. Some of these vulnerability mitigation schemes, such as the PaX Linux kernel patch, may be applied even to unmodified legacy binaries. Others can be more difficult to adopt, for example when they require complex source-code analysis.

Unfortunately, the reliance on secret values represents a vulnerability, because the values may not remain secret. In practice, a lucky, knowledgeable, or determined attacker can defeat these schemes (see [49, 51, 56]).

Secure Machine-Code Interpreters. Program Shepherding employs an efficient machine-code interpreter for implementing a security enforcement mechanism [29], as does Strata [47]. The apparent complexity of these interpreters may affect their trustworthiness and complicate their adoption. Their performance overhead may be another obstacle to their use (see Section 4.2).

On the other hand, a broad class of security policies can be implemented by a machine-code interpreter. Program Shepherding has been used, in particular, for enforcing a policy that includes certain runtime restrictions on control flow. That policy is not CFI, as we define it, but CFI could be enforced by having the interpreter implement the new instructions presented below in Section 3.1.

Other Research on Intrusion Detection. CFI is also related to a line of research on intrusion detection where a security policy for a program is derived from an inspection of the program itself or its executions [6, 18, 19, 21, 24, 25, 31, 48, 58, 59]. This security policy may be enforced at runtime using an isolated operating system mechanism, which cannot be circumvented or subverted, and which disallows invalid behaviors. The behaviors in question are often limited to sequences of system calls or library calls.

In particular, Dean and Wagner describe an intrusion-detection technique that relies on a program's static CFG to achieve "a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms" at the system-call level [58]. Most recent work in this area aims to make the security policy more precise, reducing the number of false negatives, both by making use of runtime information about function calls and returns, and also by operating at the level of library functions as well as that of system calls.

The desired precision poses efficiency and security challenges. For instance, at the time of a system call, the information contained in the user-level call stack can enable context-sensitive policies and therefore can enhance precision, but it is unreliable (as it is under program control), and maintaining a protected representation of the stack in the kernel is expensive. In this and other examples, there is a tension between efficiency and security.

CFI enforcement can be regarded as a fine-grained intrusion-detection mechanism based on a nondeterministic finite automaton. When CFI is coupled with a protected shadow call stack, the level of precision increases [18, 24]. Like previous work, CFI enforcement has difficulty with data-driven impossible paths. CFI precision is also affected by the degree of fan-in/fan-out at choice points. (The literature contains several measurements of fan-in/fan-out in program code, which we do not repeat in this paper.) Unlike previous work, on the other hand, CFI enforcement restricts the behavior of every machine code instruction in subject programs (cf. [25, Section 8]).

At the same time, CFI enforcement can be regarded as a basis for other intrusion-detection machinery. By using CFI and SMAC, that other intrusion-detection machinery could be implemented without modifications to the underlying operating system, or the cost of operating-system interactions, without a substantial loss in the level of protection or runtime overhead.

Language-Based Security. Virtually all high-level languages have execution models that imply some properties about the expected control flows. Even unsafe high-level languages are not meant to permit jumps into the middle of a machine-code instruction. Safer high-level languages, such as Java and C#, provide stronger guarantees. Their type systems, which aim to ensure memory safety, also constrain what call sequences are possible. Unfortunately, such guarantees hold only at the source level. Language implementations may not enforce them, and native method implementations may not respect them.

Similar guarantees can be obtained at the assembly and binary levels through the use of proof-carrying code (PCC) or typed assembly language (TAL) [36, 38]. Again, while PCC and TAL primarily aim to provide memory safety, they also impose static restrictions on control flow. Their properties have often been analyzed formally. The analyses focus on a model in which data memory may be modified by the subject program, but they typically do not give guarantees if another entity or a flaw may corrupt data memory (e.g., [26]).

In the long term, CFI enforcement may have a narrower set of possible benefits than the use of PCC and TAL. On the other hand, in many circumstances, CFI enforcement may be easier to adopt. CFI enforcement also addresses the need for mitigations to vulnerabilities in existing code. Finally, CFI enforcement is significantly simpler (and therefore potentially more trustworthy) than many alternative, language-based techniques, such as TAL typechecking.

2.2 CFI and Fault-Tolerance

Our work is also related to research on fault-tolerance of computer systems against soft faults (single-event upsets). Most relevant are methods that attempt to discern program execution deviation from a prescribed static CFG solely through software-based methods (e.g., [40, 45, 57]). Those methods exhibit many similarities with CFI enforcement, but also significant differences.

The main differences between CFI and these fault-tolerance approaches stem from the differences between their attack and failure models. The fault-tolerance work is focused on one-time random bit-flipping in program state and, in particular, on such bit-flipping in registers; other memory is assumed to use error-correcting codes. CFI, on the other hand, is concerned with a persistent, adversarial attacker that can arbitrarily change data memory (in particular, by exploiting program vulnerabilities), but makes certain assumptions on register contents. Most fault-tolerance work provides probabilistic guarantees whereas CFI entails that even a motivated, powerful adversary can never execute even one instruction outside the legal CFG. On the other hand, CFI does not aim to provide fault tolerance.

Most similar to CFI is the method of Oh et al. [40] and how it restricts control flow through inlined labels and checks. That method, like ours, encodes the CFG (or an approximation) by embedding a set of static, immediate bit patterns in the program code. However, in that method, the runtime checks are evaluated at the destinations

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

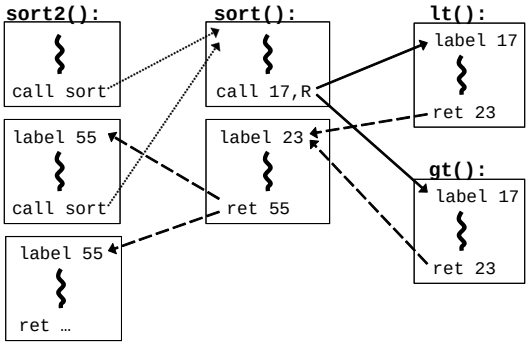

Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

of all branches and jumps, not at their sources. These checks are therefore ill-suited for our purposes. For instance, they fail to prevent jumps into the middle of functions, in particular jumps that may bypass security checks (such as access control checks). These details are consistent with the probabilistic failure model, but they would be unsatisfactory for security enforcement.

3. INLINED CFI ENFORCEMENT

As noted in the introduction, we rely on dynamic checks for enforcing CFI, and implement the checks by machine-code rewriting. We also rely on simple static inspection for verifying the correctness of this rewriting, as well for establishing other CFI properties. This section describes the basics of inlined CFI enforcement and some of its details.

Depending on the context, such as the operating system and software environment, some security enforcement mechanisms that look attractive may, in practice, be either difficult to adopt or easy to circumvent. We therefore consider not only the principles but also practical aspects of CFI enforcement, in this section and the rest of the paper.

3.1 Enforcing CFI by Instrumentation

CFI requires that, during program execution, whenever a machine-code instruction transfers control, it targets a valid destination, as determined by a CFG created ahead of time. Since most instructions target a constant destination, this requirement can usually be discharged statically. However, for computed control-flow transfers (those whose destination is determined at runtime) this requirement must be discharged with a dynamic check.

Machine-code rewriting presents an apparently straightforward strategy for implementing dynamic checks. It is however not without technical wrinkles. In particular, a rewritten program no longer uses the same code memory, and all memory addresses in the program must be adjusted accordingly. Furthermore, changes like that of the memory layout may not be possible without potentially affecting the semantics of some unconventional programs. Modern tools for binary instrumentation address these and other wrinkles, often trading generality and simplicity for efficiency [52, 53]. As a result, machine-code rewriting is practical and dependable.

It remains to design the dynamic checks. Next we explain one possible set of dynamic checks. Some of the initial explanations are deliberately simplistic, for the purposes of the exposition; variants and elaborations appear below. In particular, for these initial explanations, we rely on new machine-code instructions, with an immediate operand `ID`: an effect-free `label ID` instruction; a call instruction `call ID, DST` that transfers control to the code at the address contained in register `DST` only if that code starts with `label ID`; and a corresponding return instruction `ret ID`. Such instructions could perhaps be added to common processors to form the basis for attractive hardware CFI implementations, and should deserve further consideration. However, it is unrealistic to expect the deployment of hardware CFI support in the near future. In the remainder of the paper, we discuss only software CFI implementations. As we demonstrate, inlined CFI enforcement can be implemented in software on current processors, in particular on the x86 processor, with only a modest overhead.

CFI instrumentation modifies—according to a given CFG—each *source* instruction and each possible *destination* instruction of computed control-flow transfers. Two destinations are *equivalent* when the CFG contains edges to each from the same set of sources. For the present purposes, let us assume that if the CFG contains edges to two destinations from a common source, then the destinations are equivalent; we reconsider this assumption in Section 3.4. At each destination, instrumentation inserts a bit pattern, or ID, that identifies an equivalence class of destinations. Instrumentation also inserts, before each source, a dynamic check, or ID-check, that ensures that the runtime destination has the ID of the proper equivalence class.

Figure 1 shows a C program fragment where the function `sort2` calls a qsort-like function `sort` twice, first with `lt` and then with `gt` as the pointer to the comparison function. The right side of Figure 1 shows an outline of the machine-code blocks for these four functions and all CFG edges between them. In the figure, edges for direct calls are drawn as light, dotted arrows; edges from source instructions are drawn as solid arrows, and return edges as dashed arrows. In this example, `sort` can return to two different places in `sort2`. Therefore, the CFI instrumentation includes two IDs in the body of `sort2`, and an ID-check when returning from `sort`, arbitrarily using `55` as the ID bit pattern. (Here, we do not specify to which of the two callsites `sort` must return; Section 5 shows how to guarantee that each return goes to the most recent callsite, by using a protected shadow call stack.) Similarly, because `sort` can call either `lt` or `gt`, both comparison functions start with the ID `17`; and the `call` instruction, which uses a function pointer in register `R`, performs an ID-check for `17`. Finally, the ID `23` identifies the block that follows the comparison callsite in `sort`, so both comparison functions return with an ID-check for `23`.

This example exposes patterns that are typical when CFI instrumentation is applied to software compiled from higher-level programming languages. CFI instrumentation does not affect direct function calls: only indirect calls require an ID-check, and only functions called indirectly (such as virtual methods) require the addition of an ID. Function returns account for many ID-checks, and an ID must be inserted after each function callsite, whether that function is called indirectly or not. The remaining computed control flow is typically a result of switch statements and exceptions

Source				Destination		
Opcode bytes	Instructions			Opcode bytes	Instructions	
FF E1	jmp	ecx	; computed jump	8B 44 24 04 ...	mov eax, [esp+4]	; dst
can be instrumented as (a):						
81 39 78 56 34 12 75 13 8D 49 04 FF E1	cmp jne lea jmp	[ecx], 12345678h error_label ecx, [ecx+4] ecx	; comp ID & dst ; if != fail ; skip ID at dst ; jump to dst	78 56 34 12 8B 44 24 04 ...	; data 12345678h mov eax, [esp+4]	; ID ; dst
or, alternatively, instrumented as (b):						
B8 77 56 34 12 40 39 41 04 75 13 FF E1	mov inc cmp jne jmp	eax, 12345677h eax [ecx+4], eax error_label ecx	; load ID-1 ; add 1 for ID ; compare w/dst ; if != fail ; jump to label	3E 0F 18 05 78 56 34 12 8B 44 24 04 ...	prefetchnta [12345678h] mov eax, [esp+4]	; label ; ID ; dst

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

and, in both cases, an ID is needed at each possible destination and an ID-check at the point of dispatch.

3.2 CFI Instrumentation Code

Refining the basic scheme for CFI instrumentation, we should choose specific machine-code sequences for ID-checks and IDs. The choice is far from trivial. Those code sequences should use instructions of the architecture of interest, and ideally they should be both correct and efficient.

Figure 2 shows example x86 CFI instrumentation with two alternative forms of IDs and ID-checks, along with their actual x86 opcode bytes. The figure uses as the ID the 32-bit hexadecimal value `12345678`. The source (on the left) is a computed jump instruction `jmp ecx`, whose destination (on the right) may be a `mov` from the stack. Here, the destination is already in `ecx` so the ID-checks do not have to move it to a register—although, in general, ID-checks must do this to avoid a time-of-check-to-time-of-use race condition [8]. The code sequences for ID-checks overwrite the x86 processor flags and, in (b), a register is assumed available for use; Section 4 explains why this behavior is reasonable.

In alternative (a), the ID is inserted as data before the destination `mov` instruction, and the ID-check modifies the computed destination using a `lea` instruction to skip over the four ID bytes. The ID-check directly compares the original destination with the ID value. Thus, the ID bit pattern is embedded within the ID-check `cmp` opcode bytes. As a result, in (a), an attacker that can somehow affect the value of the `ecx` register might be able to cause a jump to the `jne` instruction instead of the intended destination.

Alternative (b) avoids the subtlety of (a), by using ID-1 as the constant in the ID-check and incrementing it to compute the ID at runtime. Also, alternative (b) does not modify the computed jump destination but, instead, inserts an effective `label ID` at the start of the destination—using a side-effect-free x86 prefetch instruction to synthesize the `label ID` instruction.

Section 4 describes machine-code sequences that build on these two alternatives.

3.3 Assumptions

In our design, CFI enforcement provides protection even against powerful adversaries that control the data memory of the executing program. The machine-code instruction sequences that implement ID-checks and IDs do not rely on the integrity of data memory. It is however critical that three assumptions hold. These three assumptions are:

UNQ Unique IDs: After CFI instrumentation, the bit patterns chosen as IDs must not be present anywhere in the code memory except in IDs and ID-checks. This property is easily achieved by making the space of IDs large enough (say 32-bit, for software of reasonable size) and by choosing IDs so that they do not conflict with the opcode bytes in the rest of the software.

NWC Non-Writable Code: It must not be possible for the program to modify code memory at runtime. Otherwise, an attacker might be able to circumvent CFI, for example by causing the overwriting of an ID-check. NWC is already true on most current systems, except during the loading of dynamic libraries and runtime code-generation.

NXD Non-Executable Data: It must not be possible for the program to execute data as if it were code. Otherwise, an attacker could cause the execution of data that is labeled with the expected ID. NXD is supported in hardware on the latest x86 processors, and Windows XP SP2 uses this support to enforce the separation of code and data [35]. NXD can also be implemented in software [41]. By itself (without CFI), NXD thwarts some attacks, but not those that exploit pre-existing code, such as "jump-to-`libc`" attacks (see Section 4.3).

Somewhat weaker assumptions may sometimes do. In particular, even without NXD, inlined CFI enforcement may be successful as long as the IDs are randomly chosen from a sufficiently large set; then, if attackers do not know the particular IDs chosen, ID-checks will probably fail whenever data execution is attempted. This "probabilistic" defense is similar to that provided by StackGuard and other mitigation mechanisms based on secrets [13, 14, 62]. Since a lucky, persistent, or knowledgeable attacker will still succeed [49, 51, 56], we do not favor this CFI variant and do not discuss it further. We believe that CFI should be supported by either hardware or software NXD; Section 5 shows how CFI enforcement can be integrated with one particular software implementation of NXD.

The assumptions can be somewhat problematic in the presence of self-modifying code, runtime code generation, and the unanticipated dynamic loading of code. Fortunately, most software is rather static—either statically linked or with a statically declared set of dynamic libraries. For example, although the Apache web server is a complex, extensible software system, configuration files bound

the set of its loadable modules prior to the start of execution. Similarly, for the Outlook email client, the Windows registry bounds the set of loadable components. Nevertheless, we are working on expanding inlined CFI enforcement with the goal of handling runtime code generation and other dynamic additions of code.

The implementation of IDs and ID-checks relies on a few registers, and requires that the values contained in those registers are not subject to tampering. This requirement is compatible with kernel-based multi-threading, since one program thread cannot affect the registers of other program threads. Furthermore, this requirement is straightforwardly met, as long as preemptive user-level context switching does not read those register values from data memory, and as long as the program in question cannot make system calls that arbitrarily change system state. This restriction on system calls is necessary for excluding system calls that make data memory executable—in contradiction with NXD—and that change code memory—in contradiction with NWC and possibly also in violation of UNQ.[1]

In general, assumptions are often vulnerabilities. When assumptions are invalidated somehow, security guarantees are diminished or void. It is therefore important to justify assumptions (as we do for NXD, for instance) or at the very least to make them explicit, to the extent possible. Of course, we recognize that, in security, any set of assumptions is likely to be incomplete. We focus on the assumptions that we consider most relevant on the basis of analysis and past experience, but for example neglect the possibility that transient hardware faults might affect instruction semantics in arbitrary ways (e.g., [40, 45, 57]).

3.4 On Destination Equivalence

Preferably, control-flow enforcement should be as precise as possible. Without some care, schemes based on IDs and ID-checks may be more permissive than necessary.

Section 3.1 assumes that if the CFG contains edges to two destinations from a common source, then the destinations are equivalent. This assumption need not always hold. For instance, in a program compiled from a language with subtyping, one may have a type T and a supertype T' that both implement a method `toString`; a `toString` invocation on T may have a single destination m while a `toString` invocation on T' may have the destination m but also a second destination m'. In this case, m and m' are not equivalent, but an imprecise CFI enforcement technique may allow control to flow from a `toString` invocation on T to m'.

One strategy for increasing precision is code duplication. For instance, two separate copies of the function `strcpy` can target two different destination sets when they return. In general, code duplication can be used for eliminating the possibility of overlapping but different destination sets. (Specifically, we can prove that a simple construction that splits CFG nodes into multiple nodes always yields graphs in which overlapping destination sets are identical.) This approach, in the limit, amounts to complete function inlining, apart from recursive calls; it has been used in several intrusion detection implementations (e.g., [25]).

[1]Most software security enforcement mechanisms adopt restrictions of this sort even for single-threaded programs, since system calls that arbitrarily change system state invalidate many assumptions of those mechanisms, and can even turn off those mechanisms. Nevertheless, the restrictions are usually left unstated because, in practice, they are difficult to satisfy without support from the operating system. CFI makes it easier to enforce the restrictions, by allowing system calls and their arguments to be constrained without any operating system modification (as discussed further in Section 5).

Alternatively, refining the instrumentation is also a good option for increasing precision. For example, more than one ID can be inserted at certain destinations, or ID-checks can sometimes compare against only certain bits of the destination IDs.

Of course, the assumption of Section 3.1 can also be made true by adding edges to the CFG, thus losing precision. In practice, this alternative can often be satisfactory: even a coarse CFI instrumentation with only one ID value—or with one ID value for the start of functions and another ID value for valid destinations for function returns—will yield significant guarantees. For instance, that instrumentation will prevent jumps into the middle of functions, which are necessary for some exploits.

3.5 Phases of Inlined CFI Enforcement

Inlined CFI enforcement can proceed in several distinct phases. The bulk of the CFI instrumentation, along with its register liveness analysis and other optimizations, can be separated from the CFG analysis on which it depends, and from install-time adjustments and verifications.

The first phase, the construction of the CFGs used for CFI enforcement, may give rise to tasks that can range from program analysis to the specification of security policies. Fortunately, a practical implementation may use standard control-flow analysis techniques (e.g., [2, 4, 58]), for instance at compile time. Section 4 describes how our x86 implementation applies these techniques by analyzing binaries (rather than source code).

After CFI instrumentation (perhaps at installation time), another mechanism can establish the UNQ assumption. Whenever software is installed or modified, IDs can be updated to remain unique, as is done with pre-binding information in some operating systems [3].

Finally (for example, when a program is loaded into memory and assembled from components and libraries), a CFI verification phase can statically validate direct jumps and similar instructions, the proper insertion of IDs and ID-checks, and the UNQ property. This last verification step has the significant benefit of making the trustworthiness of inlined CFI enforcement be independent of the complexity of the previous processing phases.

4. A PRACTICAL CFI IMPLEMENTATION

This section reports on our implementation of inlined CFI enforcement, and on measurements and experiments.

4.1 The Implementation

We have implemented inlined CFI enforcement for Windows on the x86 architecture. Our implementation relies on Vulcan [52], a mature, state-of-the art instrumentation system for x86 binaries that requires neither recompilation nor source-code access. This system addresses the challenges of machine-code rewriting in a practical fashion—as evidenced by its regular application to software produced by Microsoft. Thereby, despite being only a prototype, our implementation of inlined CFI enforcement is both practical and realistic.

Our implementation uses Vulcan for building a CFG of the program being instrumented. This CFG construction correctly handles x86 instructions that perform computed control transfers—including function returns, calls through function pointers, and instructions emitted for switch statements and other dynamic dispatch (like C++ vtables). Our CFG is conservative in that each computed `call` instruction may go to any function whose address is taken; we discover those functions with a flow-insensitive analysis of relocation entries in the binary. Our implementation is simplified by certain Windows particulars: there are no signals like those of Unix, and Windows binaries provide a "SafeSEH" static list of

Function Call		Function Return	
Opcode bytes	Instructions	Opcode bytes	Instructions
FF 53 08	call [ebx+8] ; call fptr	C2 10 00	ret 10h ; return

are instrumented using `prefetchnta` destination IDs, to become

8B 43 08	mov eax, [ebx+8] ; load fptr	8B 0C 24	mov ecx, [esp] ; load ret
3E 81 78 04 78 56 34 12	cmp [eax+4], 12345678h ; comp w/ID	83 C4 14	add esp, 14h ; pop 20
75 13	jne error_label ; if != fail	3E 81 79 04	cmp [ecx+4], ; compare
FF D0	call eax ; call fptr	DD CC BB AA	AABBCCDDh ; w/ID
3E 0F 18 05 DD CC BB AA	prefetchnta [AABBCCDDh] ; label ID	75 13	jne error_label ; if!=fail
		FF E1	jmp ecx ; jump ret

Figure 3: The CFI instrumentation of x86 `call` and `ret` used in our implementation.

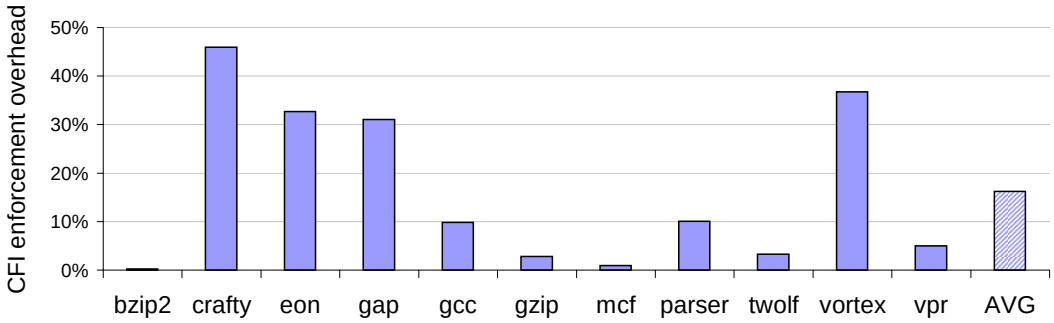

Figure 4: Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

all possible runtime exception handlers. Other CFG intricacies, such as `setjmp` and `longjmp`, are addressed using techniques from the programming-languages and the intrusion-detection literatures [4, 19, 25, 58].

Figure 3 shows how our CFI implementation rewrites the x86 machine code for an indirect function call and a corresponding function return. The destination of the `call` instruction is stored in memory at address `ebx+8`; the argument `10h` makes the `ret` instruction also pop 16 bytes of parameters off the stack. Next we explain some of the details of the rewritten code. On x86, CFI instrumentation can implement IDs in various ways (e.g., by successive opcodes that add and subtract the same constant). Our prototype, like alternative (b) of Section 3.2, uses prefetch instructions for IDs. Our ID-checks, however, take after the other alternative of Section 3.2: a `cmp` instruction directly compares against the destination ID bit pattern—and, hence, an infinite loop of the ID-check opcode bytes `3E...D0` is possible. (We do not regard such loops as a serious failure of CFI, since an attacker that controls all of memory probably has many ways of causing infinite loops.) To avoid a race condition, source instructions where the destination address resides in data memory (such as `ret`) are changed to a `jmp` to an address in a register. If an ID-check fails, our implementation immediately aborts execution by using a Windows mechanism for reporting security violations.

Our CFI instrumentation is crafted to allow low enforcement overheads for most programs. Because the IDs and ID-checks have the same locality properties as executing code, they are not penalized by high memory latency. (They may however increase pressure on caches, especially when instruction and data caches are separate.) On the x86, the ID-check instrumentation can make use of the standard calling conventions for further performance gain: in almost all cases, the `eax` and `ecx` registers can be used directly at function calls and returns, respectively, and the x86 flags do not have to be saved. During our experiments, we discovered only about a dozen functions—mostly handwritten code in standard libraries—where state (such as the x86 flags) needs to be preserved.

All CFI optimization, like the above, must be done carefully, since it can lead to a change in program semantics or to invalid CFI instrumentation. Fortunately, the use of a final verification phase can ensure that the CFI guarantees will hold during execution despite any errors in optimizations.

4.2 Measurements

We measured the overhead of our inlined CFI enforcement on some of the common SPEC computation benchmarks [54]. We performed all the experiments in this paper on Windows XP SP2 in "Safe Mode," where most daemons and kernel modules are disabled. Our hardware was a Pentium 4 x86 processor at 1.8GHz with 512MB of RAM. The target programs were compiled with Microsoft Visual C++ 7.1 using full optimizations. For SPEC, the inputs were the complete reference datasets and the output was validated as the correct result. We report the average of three runs; measurement variance was negligible, with standard deviation of less than one percent.

The CFG construction and CFI instrumentation of each binary took about 10 seconds, with the size of the binary increasing by an average 8%. Figure 4 gives the normalized overhead of CFI enforcement, shown as increase in the running time of each CFI-instrumented benchmark relative to the running time of the original benchmark binaries. The measured overhead ranged from zero to 45%, and the benchmarks took 16% longer to execute, on average.

As shown by Figure 4, our prototype inlined CFI enforcement hardly affects the performance of some programs, but it can cause a substantial slowdown of other programs. Overall, the measured performance overhead seems tolerable, even though we have not yet explored most of the optimizations possible in x86 CFI instrumentation. Because of CFI verification, such further optimization should reduce overhead without making CFI enforcement less trustworthy.

Moreover, the performance overhead of CFI enforcement is competitive with—or even better than—the cost of most comparable techniques that aim to mitigate security vulnerabilities (e.g., [13, 29, 46]). For instance, the overhead of Program Shepherding is more than 100% for the benchmark program `crafty` on Windows; the corresponding CFI enforcement overhead is 45%, and this is our highest measured overhead. Similarly, the overhead of Program Shepherding is more than 660% for `gcc` on Windows, and can be brought down to 35% only by exposing the security mechanism itself to attack; the corresponding CFI enforcement overhead is under 10%.

Note that the SPEC benchmarks focus on CPU-intensive programs with integer arithmetic. CFI will cause relatively less overhead for I/O-driven server workloads. For example, one might expect to see an even smaller performance impact on FTP than on SPEC (as in [63]).

4.3 Security-Related Experiments

It is difficult to quantify the security benefits of any given mitigation technology: the effects of unexploited vulnerabilities cannot be predicted, and real-world attacks—which tend to depend on particular system details—can be thwarted, without any security benefits, by trivial changes to those details.

Even so, in order to assess the effectiveness of CFI, we examined by hand some well-known security exploits (such as those of the Blaster and Slammer worms) as well as several recently reported vulnerabilities (such as the Windows ASN.1 and GDI+ JPEG flaws). CFI would not have prevented Nimda and some similar exploits that rely on the incorrect parsing of input strings, such as URLs, to cause the improper launch of the `cmd.exe` shell or some other dangerous executable (see also [10]). On the other hand, CFI would have prevented all the other exploits that we studied because, in one way or another, they all endeavored to deviate from the expected control flow. Many exploits performed a "jump-to-`libc`" control transfer from a program point where this jump was not expected. Often this invalid control transfer was attempted through heap overflows or some form of pointer subterfuge (of the kind recently described by Pincus and Baker [42]).

Pointer subterfuge relies on modifications to data memory, and can result in possibly arbitrary further modifications to data memory. Hence, thwarting pointer subterfuge calls for techniques that—like ours—afford protection even when attackers are in full control of data memory.

As a concrete example, let us consider the published attack on the GDI+ JPEG flaw in Windows [20]. This attack starts by causing a memory corruption, overwriting a global variable that holds a C++ object pointer. When this pointer is later used for calling a virtual destructor, the attacker has the possibility of executing code of their choice. A CFI ID-check at this callsite can prevent this exploit, for instance by restricting valid destinations to the C++ virtual destructor methods of the GDI+ library.

As another concrete example that illustrates the benefits of CFI, we discuss the following C function, which is intended to return the median value of an array of integers:

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_LEN
   int tmp[MAX_LEN];
   memcpy( tmp, data, len*sizeof(int) );
   qsort( tmp, len, sizeof(int), cmp );
   return tmp[len/2];
}
```

This code is vulnerable—and can be exploited by an attacker that controls the inputs—even on systems that use deployed mitigation techniques such as stack canaries and hardware NXD support. We have constructed actual exploits for this vulnerability; they work even on Windows XP SP2 with x86 hardware NXD support. One exploit is based on a traditional stack-based buffer overflow; others work via C++ vtables and the heap. The exploits overwrite the `qsort` comparison function pointer, pointing it to a sequence of four machine-code opcode bytes (found in the middle of an existing function) which reposition the stack pointer `esp` in a particular way. Subsequently, when `cmp` is called, the exploit proceeds through the unwinding of the stack; as each frame is popped, the exploit executes a particular piece of existing code. Eventually, the attacker has full control over the system.

With CFI, on the other hand, the vulnerability in the `median` function cannot be exploited in this manner. CFI forbids invalid control transfers into the middle of functions (as well as returns to the start of functions), and it therefore prevents the necessary first step of the exploits (and would also prevent subsequent steps). This protection is not dependent on how closely the CFI runtime guarantees correspond to a precise CFG of the program; even a coarse CFG has the desired effect.

For a final set of experiments, we ported to Windows a suite of 18 tests for dynamic buffer-overflow prevention developed by Wilander and Kamkar [61]. (Wilander and Kamkar were unable to provide us with the source code for two of the 20 tests from their study.) The tests in the original suite concern whether attackers could directly execute shellcode of their choosing. We extended the tests to consider also "jump-to-`libc`" and pointer subterfuge attacks. We computed CFGs for these tests, and applied our CFI instrumentation. With CFI enforcement, none of the exploits in this test suite were successful, because they attempted to deviate from the corresponding CFGs. This result is encouraging, especially since other mitigation techniques have failed these tests [61].

5. BUILDING ON CFI

CFI ensures that runtime execution proceeds along a given CFG, guaranteeing, for instance, that the execution of a typical function always starts at the beginning and proceeds from beginning to end. Thereby, CFI can increase the reliability of any CFG-based technique (for example, strengthening previous techniques against buffer overflows and for intrusion detection [32, 58]).

This section describes other applications of CFI, as a foundation for Inlined Reference Monitors (IRMs), for SFI in particular, and for Software Memory Access Control (SMAC), which we introduce here. It also shows how to tighten CFI enforcement by relying on either SMAC or standard x86 hardware support.

5.1 CFI as a Foundation for IRMs

IRMs enforce security policies by inserting into subject programs the code for validity checks and also any additional state that is needed for enforcement [16]. IRMs require that the subject program can neither circumvent the inserted validity checks nor subvert the added state. By constraining the CFG enforced by CFI, the first of these requirements is easily satisfied. Further, SMAC (discussed below) supports isolated data memory regions in which the added IRM state can be safely kept. Thus, CFI and SMAC greatly facilitate the implementation of IRMs.

In particular, CFI can contribute to the IRM enforcement of security policies that restrict a program's use of the underlying operating system (for instance, preventing files with some filenames from being written) [44]. Such policies are often necessary; many of their existing implementations modify operating systems, something that CFI enables us to avoid. With CFI, it is easy to enumerate those points in a program where system calls can be made. At each

```
int compute_sum( int a[], int len )
{
    int sum = 0;
    for(int i = 0; i < len; ++i) {
        sum += a[i];
    }
    return sum;
}
```

```
        ...
        mov  ecx, 0h          ; int i = 0
        mov  esi, [esp+8]     ; a[] base ptr
        and  esi, 20FFFFFFh   ; SFI masking
LOOP:   add  eax, [esi+ecx*4] ; sum += a[i]
        inc  ecx              ; ++i
        cmp  ecx, edx         ; i < len
        jl   LOOP
```

Figure 5: Leveraging CFI for optimizations: hoisting an SFI check out of a loop.

such point, an IRM validity check can be inserted, and CFI can ensure that the check cannot be circumvented.

5.2 Faster SFI

Software Fault Isolation (SFI) is one particular type of IRM designed to emulate traditional memory protection. In SFI, code is inserted at each machine-code instruction that accesses memory to ensure that the target memory address lies within a certain range [17, 34, 50, 60].

Much as in Section 5.1, CFI makes SFI instrumentation non-circumventable. CFI can also reduce SFI overhead. For instance, the guarantees about control flow remove the need to check memory addresses in local variables repeatedly. Figure 5 demonstrates one such optimization. The figure shows a C function that sums the contents of an array, and the first two basic blocks of the x86 machine code that a compiler might emit for this function. (The start of the first basic block is elided.) The machine code includes an `and` instruction that masks off the top bits from the base address of the array, constraining the array to reside at an address whose top eight bits are either `00h` or `20h`. As long as the low memory (whose addresses start with `00h`) is inaccessible, this use of an `and` instruction can establish several disjoint, isolated memory regions as demonstrated in PittSFIeld, a recent, efficient x86 SFI implementation [34].

The SFI literature is full of other optimizations that simplify the inserted checks. For example, checks can often be eliminated when memory is accessed through a register plus a small constant offset, as long as inaccessible "guard pages" are placed before and after permitted memory ranges. This optimization is especially useful for accesses to local, stack variables, such as reading the value at `esp+8` in Figure 5. However, the weak control-flow guarantees of past SFI implementations make it difficult to reason about program behavior and, partly as a result, past optimizations have sometimes led to vulnerabilities [17, 34].

CFI makes optimizations more robust and enables many new ones. For the code in Figure 5, CFI allows the `and` instruction to be hoisted out of the loop; thus, at runtime, a single masking operation suffices for checking all memory accesses due to the array. Past implementations of SFI require a masking operation to accompany each execution of the `add` instruction, because a computed jump might result in that instruction executing with arbitrary values in registers `esi` and `ecx`. CFI precludes such computed jumps, and with CFI it is easy see that loop execution does not change `esi` and increments `ecx` from a base of zero.

These optimization can result in a striking overhead reduction. The SFI literature includes measurements of three systems for x86: Vino's MiSFIT, x86 SASI, and PittSFIeld [17, 34, 50]. For comparison, we applied CFI and SFI, with the optimizations of Figure 5, to two benchmark programs, `hotlist` and the C reference implementation of MD5. The `hotlist` benchmark searches a linked list of integers for a particular value [50]. For `hotlist`, MiSFIT and SASI produce 140% and 260% overheads, respectively, when both memory reads and writes are restricted. Our corresponding measurement shows only 22% overhead. For MD5, the reported performance overheads for PittSFIeld and MiSFIT range from 23% to 50% [34, 50]. Our corresponding measurement shows only 4.7% overhead.

For this preliminary investigation of SFI, we performed some of the machine-code rewriting by hand on the two benchmark programs. As is common in previous work on SFI, we also made several simplifying assumptions about memory layouts, for example that low memory is inaccessible. In many existing systems, those assumptions cannot be fully satisfied. For useful, realistic memory protection, the rewriting process should be fully automated, and those assumptions should be removed. The SFI policy should also be revisited. We are currently developing a system that addresses these concerns, in collaboration with Michael Vrable.

5.3 SMAC: Generalized SFI

SMAC is an extension of SFI that allows different access checks to be inserted at different instructions in the program being constrained. Therefore, SMAC can enforce policies other than those of traditional memory protection. In particular, SMAC can create isolated data memory regions that are accessible from only specific pieces of program code, for instance, from a library function or even individual instructions. Thus, SMAC can be used to implement security-relevant state for IRMs that cannot be subverted. For instance, the names of files about to be opened can first be copied to memory only accessible from the "FileOpen" function, and then checked against a security policy.

CFI can help with SMAC optimizations, much as it does for SFI optimizations; conversely, SMAC can help in eliminating CFI assumptions. SMAC can remove the need for NWC, by disallowing writes to certain memory addresses, and for NXD, by preventing control flow outside those addresses. (This synergy between CFI and SMAC is not a circular-reasoning fallacy, as we demonstrate in the formal treatment of CFI with SMAC [1].)

Figure 6 shows SMAC instrumentation that can guarantee that only code is executed. As in Figure 5, an `and` instruction masks off the top bits of the destination addresses of computed x86 function calls and returns. Thus, code memory is restricted to addresses whose top eight bits are `40h` (provided that addresses that start with `00h` are invalid). To ensure NWC and NXD for simple regions of code, stack, and data, the SMAC checks can be as straightforward as this single `and` instruction.

Alternatively, the SMAC checks might embody elaborate policies, and allow arbitrary layouts of data and code memory regions, although the code for such checks is likely to be more complex and less efficient than that of Figure 6. In this paper, since it suffices for our immediate purposes, we follow the SFI literature and focus on coarser-grained memory protection.

5.4 A Protected Shadow Call Stack

Because CFI concerns a finite, static CFG, it cannot ensure that a function call returns to the callsite most recently used for invoking the function. Complementing CFI with the runtime call stack

```
call  eax            ; call func ptr          ret                ; return
```

with CFI, and SMAC discharging the NXD requirement, can become:

```
and  eax, 40FFFFFFh  ; mask func ptr          mov  ecx, [esp]      ; load return dst
cmp  [eax+4], ID     ; compare dst w/ID       and  ecx, 40FFFFFFh  ; mask return dst
jne  error_label     ; if != fail             cmp  [ecx+4], ID     ; comp dst w/ID
call eax             ; call func ptr          jne  error_label     ; if != fail
prefetchnta ID       ; label ID               add  esp, 4h         ; pop 4
                                              jmp  ecx             ; jump return dst
```

Figure 6: Instrumentation of x86 `call` and `ret`, with CFI and SMAC.

```
call  eax            ; call func ptr          ret                ; return
```

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

```
add  gs:[0h], 4h     ; inc stack by 4         mov  ecx, gs:[0h]    ; get top offset
mov  ecx, gs:[0h]    ; get top offset         mov  ecx, gs:[ecx]   ; pop return dst
mov  gs:[ecx], LRET  ; push ret dst           sub  gs:[0h], 4h     ; dec stack by 4
cmp  [eax+4], ID     ; comp fptr w/ID         add  esp, 4h         ; skip extra ret
jne  error_label     ; if != fail             jmp  ecx             ; jump return dst
call eax             ; call func ptr
LRET: ...
```

Figure 7: Instrumentation of x86 `call` and `ret`, with CFI and a protected shadow call stack.

(see [11, 22, 23, 24, 37, 43]) can guarantee this property and increase the precision of CFI enforcement. However, if CFI is to rely on runtime information such as a call stack, the information should not be maintained in unprotected memory, as the ordinary call stack usually is, since the attacker may corrupt or control unprotected memory. Therefore, a protected shadow call stack is required. The assumption that attackers cannot modify this stack directly is necessary, but not sufficient. It is also crucial to guard the stack against corruption that may result from program execution.

One possible strategy for implementing a protected shadow call stack employs CFI and SMAC. Specifically, the shadow call stack may be maintained in a memory region whose addresses start with a specific prefix (e.g., `10h`), and protected by SMAC checks such as those of Section 5.3. Static verification can then ensure that only SMAC instrumentation code at call and return instructions can modify this memory region, and only by correctly pushing and popping the correct values.

In this section we focus on an alternative implementation strategy. The resulting implementation is even simpler and more efficient than one that employs SMAC. It leverages the CFI guarantees and standard x86 hardware support. Specifically, we maintain the shadow call stack in an isolated x86 segment[2]. With CFI, static verification can ensure that a particular segment register, or segment selector, is used properly by the instrumentation code for call and return instructions, and that only this instrumentation code accesses the corresponding segment. Without CFI, on the other hand, it is extremely difficult to trust the use of segments in an x86 machine-code sequence of non-trivial size. For instance, the opcodes for loading an improper segment selector might be found within basic blocks in system library code, or even within the opcodes of a long, multi-byte instruction; without CFI, an attacker might be able to direct execution to those places.

Figure 7 shows how we use segments in our instrumentation. The segment register `gs` always points to the memory segment that holds the shadow call stack and which has been created to be isolated and disjoint from other accessible memory segments. On Windows, `gs` is unused in application code; therefore, without limitation, CFI verification can statically preclude its use outside this instrumentation code. As shown in the figure, the instrumentation code maintains (in memory location `gs:[0h]`) an offset into this segment that always points to the top of the stack. The use of the protected shadow call stack implies that each return goes to the correct destination, so no ID-checks are required on returns in this instrumentation code.

The isolated memory segment for the shadow call stack can be created by user-mode application code, as long as this activity happens before all other code executes, and only this code loads new selectors into segment registers. For each thread of execution, this initial code can truncate the existing code and data segments and specify that the new, isolated segment lies within the freed-up address region. CFI can guarantee that the machine code for this setup activity will remain inaccessible once it has executed.

Alternatively, the isolated memory segment might be created by the operating system. Support from the operating system could provide other benefits, such as reduced resource consumption by fast detection of overflows in the shadow call stack (for example, using "guard pages") and dynamic increases in the segment size. We do not assume this support from the operating system, as it is not standard at present. We depend only on current Windows features.

We have implemented a protected shadow call stack for Windows on the x86 architecture, relying on segments and CFI, as outlined above. Figure 8 shows detailed performance measurements for the SPEC benchmarks. We observed only a modest performance overhead for our CFI-protected shadow call stack instrumentation: on average 21%, with 5% for `gzip` and 11% for `gcc`. The overhead includes that of CFI enforcement without the unnecessary ID-checks on returns. These measurements are consis-

[2]The x86 architecture allows multiple *segments* to exist simultaneously within an application. A segment is a specified region of memory, named using an ordinal *selector*. A segment is adopted by loading its ordinal into a *segment register*; there are six such registers, of which some are rarely, if ever, used in modern application code. All memory accesses are performed relative to a segment specified by a segment register; the instructions determine which segment register is to be used, either implicitly or explicitly. On most popular operating systems, user-level code can specify memory regions for its own local segments, which are then context-switched with the application.

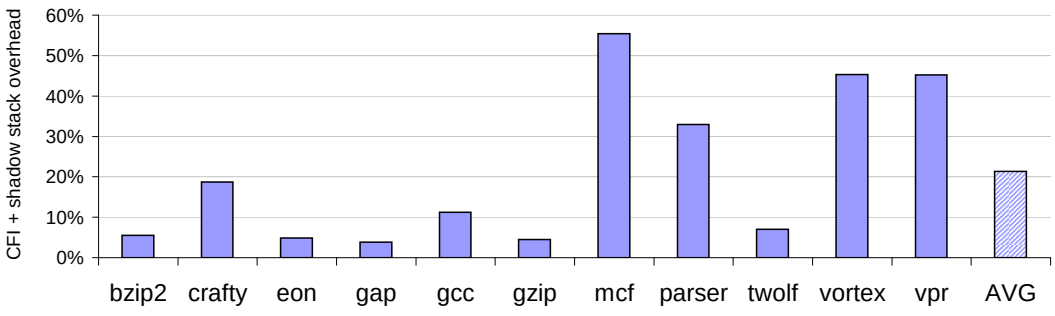

Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.

tent with the overhead reported in the literature for *unprotected* shadow call stacks (whose integrity could be subverted by attackers) [43]. In contrast, the overhead reported in the literature for *protected* shadow call stacks ranges from 729% (for `gzip`) to 1900% (for `gcc`) [11, 23]. While the dramatic improvement that we obtain is partly due to the use of segments, this use of segments is possible only because of CFI.

Once we have a protected shadow call stack, further restrictions on control flow become possible. For example, the control-flow policy could require that every call from a certain function `g` to another function `h` be immediately preceded by a call from a third function `f` to `g`. (Analogous restrictions often appear in the literature on intrusion detection.) Even further restrictions become possible if we keep a protected computation history that records all control transfers. For example, the control-flow policy could then require that a certain function `f` is called at most as often as another function `g`. Such restrictions may sometimes be desirable; for instance, they might prevent some "confused deputy" attacks [28]. On the other hand, we believe that even the simplest kind of CFI enforcement is quite effective at thwarting external attacks that aim to control software behavior.

6. FORMAL STUDY (SUMMARY)

In this section we summarize our formal study of CFI. Another paper contains further details [1]. We view this study as central to our work, as a major difference with literature on previous mitigation tools, and as an important similarity with research on type-safe languages. We have found it helpful for clarifying hypotheses and guarantees. We have also found it helpful as a guide: in our research, we rejected several techniques that were based on unclear assumptions or that would have provided hard-to-define protections.

More specifically, this study includes a semantics for programs, definitions for program instrumentation, and results about the behavior of instrumented programs. The semantics allows for the possibility that an attacker controls data memory. The program instrumentation has two variants, with and without SMAC; the latter addresses a machine model with weaker assumptions. In what follows, we focus on the variant without SMAC, except where otherwise noted. Our main theorems establish that CFI holds for instrumented programs.

The machine model and the programs that we employ are typical of research in programming-language theory. They enable us to consider CFI but exclude virtual memory, dynamic linking, threading, and other sophisticated features found in actual systems. In the machine model, an execution state consists of a code memory M_c, a data memory M_d, an assignment of values to registers R, and a program counter pc. Here, M_c and M_d map addresses to words, R maps register numbers to words, and pc is a word. Essentially, our language is a minor variant of that of Hamid et al. [27]. We add only an instruction in which an immediate value can be embedded, as a label, and which behaves like a nop. It is directly analogous to the `label ID` instruction of Section 3.1.

We give a formal operational semantics for the instructions of our language. For each of the instructions, the semantics says how the instruction affects memory, the machine registers, and the program counter. For example, for the instruction $add\ r_d, r_s, r_t$, the semantics says:

> If $M_c(pc)$ contains the encoding of $add\ r_d, r_s, r_t$, and the current state has code memory M_c, data memory M_d, program counter value pc, and register values R, and if $pc + 1$ is within the domain of M_c, then in the next state the code memory and data memory are still M_c and M_d, respectively, pc is incremented, and R is updated so that it maps r_d to $R(r_s) + R(r_t)$.

We consider SMAC with a variant of this semantics that includes fewer built-in checks. In the example of the $add\ r_d, r_s, r_t$ instruction, the variant does not include the precondition that $pc + 1$ is within the domain of M_c. In other words, the machine model allows the possibility that pc points outside code memory, and the instrumentation aims to ensure that this possibility is harmless.

We depart significantly from the work of Hamid et al. and other previous work by including a representation of the attacker in our model. Despite its simplicity, we regard this departure as one of our main formal contributions. Since the attacker that we have in mind is quite powerful, one might imagine that it could be difficult to capture all its capabilities. Fortunately, we can adopt an economical representation of the attacker. This representation consists in introducing one more rule into our operational semantics. The new rule expresses attacker steps, and says that at any time the attacker may modify data memory and most registers. It excludes the small number of distinguished registers on which the instrumentation relies; it also excludes code memory, consistently with our assumption NWC.

As usual in programming-language theory, the operational semantics describes state transitions by precise rules. For the instruction $add\ r_d, r_s, r_t$, for example, we have that:

$$(M_c|M_d, R, pc) \rightarrow_n (M_c|M_d, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc + 1)$$

when $M_c(pc)$ holds $add\ r_d, r_s, r_t$ and $pc + 1$ is in the domain of M_c. The relation $\rightarrow_n$ is a binary relation on states that expresses normal execution steps. For the attacker, we have a rule that enables

the following transitions, for all M_c, M_d, $M_d{}'$, R, and pc:

$$(M_c|M_d, R, pc) \rightarrow_a (M_c|M_d{}', R, pc)$$

The relation $\rightarrow_a$ is a binary relation on states, and $M_d{}'$ is the arbitrary new value of the data memory. We do not show the modifications to registers, for simplicity—our actual rule is more general in this respect. The next-state relation $\rightarrow$ is the union of $\rightarrow_n$ and $\rightarrow_a$.

In our formal study, instrumentation is treated as a series of precise checks on programs. The checks capture the conditions that well-instrumented code should satisfy, and do not address how the instrumentation happens. Only the former concern is directly relevant to security. We write $I(M_c)$ when code memory M_c is well-instrumented according to our criteria. These criteria include, for example, that every computed jump instruction is preceded by a particular sequence of instructions, which depends on a given CFG. When we consider SMAC, we also require that every memory operation is preceded by a particular sequence of instructions. Those sequences are analogous to the ones used in our actual implementation and described in detail in this paper. There are however differences in specifics, largely because of the differences between the simple machine model of our formal study and the x86 architecture.

With these definitions, we obtain formal results about our instrumentation methods. Those results express the integrity of control flow despite memory modifications by an attacker. Our main theorems say that every execution step of an instrumented program is either an attack step in which the program counter does not change, or a normal step to a state with a valid successor program counter. That is:

> Let S_0 be a state with code memory M_c such that $I(M_c)$ and $pc = 0$, and let $S_1, \ldots, S_n$ be states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \rightarrow_a S_{i+1}$ or the pc at S_{i+1} is one of the allowed successors for the pc at S_i according to the given CFG.

Thus, despite attack steps, the program counter always follows the CFG. The proof of these theorems consist in fairly conventional but long inductions on executions.

We have yet to pursue a similar formal study for the x86 architecture. Such a study may well be viable (as suggested by recent work [34]), but it may produce diminishing returns, and it would be arduous, not least because of the current absence of a complete formal specification for the x86 architecture.

7. CONCLUSION

The use of high-level programming languages has, for a long time, implied that only certain control flow should be expected during software execution. Even so, at the machine-code level, relatively little effort has been spent on guaranteeing that control actually flows as expected. The absence of runtime control-flow guarantees has a pervasive impact on all software analysis, processing, and optimization—and it also enables many of today's exploits.

CFI instrumentation aims to change this situation by embedding within software executables both a control-flow policy to be enforced at runtime and the mechanism for that enforcement. Inlined CFI enforcement is practical on modern processors, is compatible with most existing software, and has little performance overhead. CFI can also enable substantial performance improvements for other security mechanisms. Finally, CFI is simple, verifiable, and amenable to formal analysis, yielding strong guarantees even in the presence of a powerful adversary.

Acknowledgments. Martín Abadi and Jay Ligatti participated in this work while at Microsoft Research, Silicon Valley. Discussions with Greg Morrisett and Ilya Mironov were helpful to this paper's development and improved its exposition. Milenko Drinic and Andrew Edwards of the Vulcan team were helpful to our implementation efforts.

8. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, 2005. A preliminary version appears as Microsoft Research Technical Report MSR-TR-05-17, February 2005.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1985.

[3] Apple Computer. Prebinding notes, 2003. `http://developer.apple.com/releasenotes/DeveloperTools/Prebinding.html`.

[4] D. Atkinson. Call graph extraction in the presence of function pointers. In *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002.

[5] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *Proceedings of the Usenix Security Symposium*, pages 45–56, 2004.

[6] S. Basu and P. Uppuluri. Proxy-annotated control flow graphs: Deterministic context-sensitive monitoring for intrusion detection. In *ICDCIT: Proceedings of the International Conference on Distributed Computing and Internet Technology*, pages 353–362, 2004.

[7] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the Usenix Security Symposium*, pages 105–120, 2003.

[8] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, 1996.

[9] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the Usenix Security Symposium*, pages 57–72, 2004.

[10] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the Usenix Security Symposium*, pages 177–192, 2005.

[11] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 409–419, 2001.

[12] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the Usenix Security Symposium*, 2001.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*, pages 91–104, 2003.

[14] 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 *Proceedings of the Usenix Security Symposium*, pages 63–78, 1998.

[15] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture*, 2004.

[16] Ú. Erlingsson and F. Schneider. IRM enforcement of java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[17] Ú. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, 1999.

[18] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 194–210, 2004.

[19] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–77, 2003.

[20] E. Florio. Gdiplus vuln - ms04-028 - crash test jpeg. *full-disclosure at lists.netsys.com*, 2004. Forum message, sent September 15.

[21] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[22] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the Usenix Security Symposium*, pages 55–66, 2001.

[23] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the Usenix Security Symposium*, pages 61–79, 2002.

[24] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *NDSS '04: Proceedings of the Network and Distributed System Security Symposium*, 2004.

[25] R. Gopalakrishna, E. Spafford, and J.Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 18–31, 2005.

[26] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[27] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A Syntactic Approach to Foundational Proof-Carrying Code. Technical Report YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University, New Haven, CT, 2002.

[28] N. Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, 1988.

[29] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the Usenix Security Symposium*, pages 191–206, 2002.

[30] D. Kirovski and M.Drinic. POPI — a novel platform for intrusion prevention. In *Proceedings of the International Symposium on Microarchitecture*, 2004.

[31] L. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *RAID '04: Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2004.

[32] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*, pages 177–190, 2001.

[33] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the Usenix Security Symposium*, pages 121–136, 2003.

[34] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-LCS-TR-988, MIT Laboratory for Computer Science, 2005.

[35] Microsoft Corporation. Changes to functionality in Microsoft Windows XP SP2: Memory protection technologies, 2004. `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`.

[36] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[37] D. Nebenzahl and A. Wool. Install-time vaccination of Windows executables to defend against stack smashing attacks. In *Proceedings of the IFIP International Information Security Conference*, 2004.

[38] G. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[39] G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[40] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control flow checking by software signatures. *IEEE Transactions on Reliability*, 51(2), 2002. Special Section on: Fault Tolerant VLSI Systems.

[41] PaX Project. The PaX project, 2004. `http://pax.grsecurity.net/`.

[42] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[43] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the Usenix Technical Conference*, pages 211–224, 2003.

[44] N. Provos. Improving host security with system call policies. In *Proceedings of the Usenix Security Symposium*, pages 257–272, 2003.

[45] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[46] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of Network and Distributed System Security Symposium*, 2004.

[47] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 209, 2002.

[48] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[49] 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 ACM Conference on Computer and Communications Security*, 2004.

[50] C. Small. A tool for constructing safe extensible C++ systems. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[51] A. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proceedings of the Usenix Security Symposium*, pages 145–160, 2005.

[52] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[53] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report WRL Research Report 94/2, Digital Equipment Corporation, 1994.

[54] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmark suite, 2000. `http://www.spec.org/osg/cpu2000/`.

[55] G. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[56] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the International Symposium on Microarchitecture*, 2004.

[57] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of 9th IEEE International On-Line Testing Symposium*, 2003.

[58] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[59] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[60] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1993.

[61] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*, 2003.

[62] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In *Proceedings of the Symposium on Reliable and Distributed Systems*, 2003.

[63] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks, 2002.