# Control-Flow Integrity:
## Principles, Implementations, and Applications

Martín Abadi[1], Mihai Budiu[2], Úlfar Erlingsson[2], and Jay Ligatti[3]

[1]University of California, Santa Cruz
[2]Microsoft Research, Silicon Valley
[3]Princeton University

November 1, 2004

**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 efficiently implemented using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, such as policies that constrain the use of data memory.

## 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 makes 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 [8], runtime elimination of buffer overflows [27], randomization and artificial heterogeneity [23, 35], and tainting of suspect data [29]. 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 [34, 24, 28].

The shortcomings and failures of these mitigations stem, in part, from their lack of a realistic attack model and their 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.

This paper describes and studies one mitigation technique, the enforcement of *Control-Flow Integrity* (CFI), that aims to meet this standard for trustworthiness. The paper and? introduces CFI enforcement, develops some of its theory, presents an implementation for Windows on the x86 architecture, and suggests applications, exploring one in some detail.

The CFI security policy dictates that software execution must follow the 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 analysis of software binaries and their debugging information. CFGs can also be defined by explicit security policies, for example rules that require the mediation of a reference monitor for certain calls.

A security policy is of limited value without an attack model. In our design, CFI enforcement is expected to work even against powerful adversaries that control the data memory of the executing program. This model of adversaries may seem rather pessimistic, but it has the virtue of being clear, and allows for the possibility that buffer overflows and other vulnerabilities would lead to arbitrary changes in data memory.

Whereas CFI enforcement can potentially be done in several ways, we rely on machine-code rewriting that instruments software with runtime checks. These 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 programs on commodity systems, and yields efficient code. Although machine-code rewriting can be rather elaborate, it is simple to verify the proper use of instrumentation in order to guarantee inlined CFI enforcement. Thus, having defined instrumentation conditions, we prove the correctness of inlined CFI enforcement, formally, for an abstract machine with a simplified instruction set; this formal treatment of inlined CFI enforcement contributes to assurance and also serves as a guide in our design.

CFI enforcement is effective against a wide range of common attacks, because abnormal control-flow modification is a common first step in many exploits—independently of whether buffer overflows and other vulnerabilities are being exploited [24]. Of course, CFI enforcement is not a panacea: exploits within the bounds of the allowed CFG are unaffected.

No matter how the CFG is defined or how permissive it is, CFI can be a useful foundation for the enforcement of further security policies, including those that prevent higher-level attacks. In particular, CFI can prevent the circumvention of both *Inlined Reference Monitors* (IRMs) and *Software Fault Isolation* (SFI[1]) [33, 19, 17]. Further, CFI can serve as the basis of a generalization of SFI that we call *Software Memory Access Control* (SMAC). SMAC, in turn, can serve for eliminating some CFI assumptions and for enhancing CFI with runtime information that allows us to restrict control flow even more tightly.

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

[1]Both SFI and CFI instrumentation insert runtime checks before certain instructions—that is one reason for the similarity between the two acronyms.

returns [8, 25], by encrypting function pointers [35, 7], or even by interpreting software using the techniques of dynamic machine-code translation [13]. CFI enforcement aims to improve on these techniques by relying on precise definitions and offering clear, useful guarantees, with a viable, attractive implementation strategy. Finally, CFI enforcement may facilitate and complement much previous work on intrusion detection [31] and system call policies [26]. We discuss related work further in Section 7.

The next section informally explains CFI and its inlined enforcement. Section 3 presents a formal analysis of inlined CFI enforcement. Section 4 then describes SMAC and some of its applications and, in general, shows how additional security enforcement can be built on CFI. Section 5 extends the formal analysis of Section 3 with a treatment of SMAC. Section 6 describes a CFI implementation for Windows on the x86 architecture and gives performance results. Finally, Sections 7 and 8 discuss related work and draw conclusions, respectively. An Appendix contains additional details.

That work often focuses on interprocess interactions, typically with the operating systeml, and relies on the operating system for securing data; the use of our techniques may conceivably help in efficiency and generality. contents to be revised

## 2 Inlined CFI Enforcement

As noted in the introduction, we rely on dynamic checks for enforcing SFI, and implement the checks by machine-code rewriting. 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.

### 2.1 Enforcing CFI by Instrumentation

CFI requires that during program execution all control-flow machine-code instructions target a valid destination, as determined by a CFG created ahead of time. Since most instructions that affect control flow target a constant destination (encoded as an immediate operand), this requirement can usually be discharged statically. However, for computed control-flow transfers (those whose destination is determined at runtime) of the kind caused by computed-jump instructions, CFI must be enforced with dynamic checks of destination validity.

Machine-code rewriting offers an apparently straightforward strategy for implementing dynamic checks. It is however not without technical wrinkes. In particular, it implies adjusting memory addresses; a static table that maps old code addresses to new ones may be employed for redirecting all jumps dynamically, correctly but with a substantial performance penalty. Modern tools for binary instrumentation address this and other wrinkles, often trading generality and simplicity for efficiency. 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 `nop ID` instruction; a call instruction `call ID, DST` that transfers control to the

```
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 );
}
```

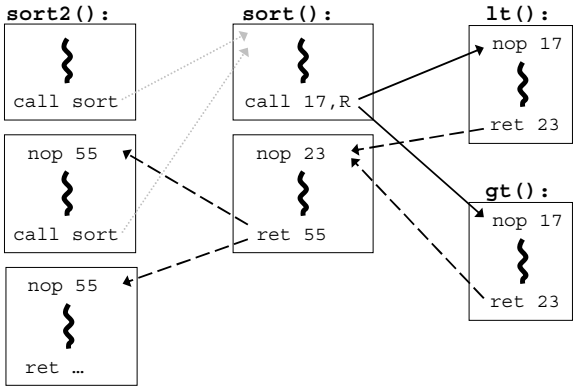


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

code at `DST` only if that code starts with `nop ID`; and a corresponding return instruction `ret ID`. Such instructions could easily be added to common processors to form the basis for a hardware CFI implementation with minimal performance overhead. As we demonstrate below, inlined CFI enforcement can also be implemented on current systems that do not include these instructions, in particular on the x86 processor.

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 them from the same sources. 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 that ensures that the runtime destination has the ID of the proper equivalence class.

alg?
value?

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 in gray; edges from source instructions are drawn in black, with return edges also dashed. 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 call-sites `sort` must return; Section 4 shows how to guarantee that each return goes to the most recent call-site, by using runtime information.) 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 following the comparison call-site in `sort`, so both comparison functions return with an ID-check for `23`.

This example exposes patterns that are typical when CFI 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 (e.g., virtual methods) require the addition of an ID. Function returns account for many ID-checks, and an ID must be inserted after each function call-site, whether that function is called indirectly or not. The remaining computed control flow is typically a result of switch

| Source | | Destination | |
|---|---|---|---|
| Opcode bytes | Instruction | Opcode bytes | Instruction |
| FF E1 | jmp ecx | 8B 44 24 04 | mov eax, [esp+4] |
| | | ... | |

can be CFI instrumented as (a):

| | | | |
|---|---|---|---|
| 81 39 | cmp [ecx], | 78 56 34 12 | ; data 12345678h |
| 78 56 34 12 | 12345678h | 8B 44 24 04 | mov eax, [esp+4] |
| 75 13 | jne error_label | ... | |
| 8D 49 04 | lea ecx, [ecx+4] | | |
| FF E1 | jmp ecx | | |

or, alternatively, CFI instrumented as (b):

| | | | |
|---|---|---|---|
| B8 | mov eax, | 3E 0F 18 05 | prefetchnta |
| 77 56 34 12 | 12345677h | 78 56 34 12 | ds:[12345678h] |
| 40 | inc eax | 8B 44 24 04 | mov eax, [esp+4] |
| 40 39 41 04 | cmp [ecx+4], eax | ... | |
| 75 13 | jne error_label | | |
| FF E1 | jmp ecx | | |

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

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

Instrumentation, such as the one sketched here, need not always preserve the semantics of programs. For example, a program whose outputs depends on code addresses may be affected. We are particularly concerned about not disrupting the operation of "everyday" programs (most ordinary machine-code programs typically produced from high-level languages by standard compilers). Even for an "everyday" program, the instrumentation can cause an execution to halt when there are attack steps, thus changing program semantics—as desired in this case. On the other hand, the instrumentation should preserve the semantics of an "everyday" program provided its control-flow graph is a conservative approximation of the normal dynamic semantics and there are no attack steps.

## 2.2 Choosing Code for CFI Instrumentation

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 ID-checks and IDs, along with their actual x86 opcode bytes. In the figure, 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. The code sequences for ID-checks overwrite the processor flags and, in (b), a register is assumed available for use; Section 6 explains why this behavior is reasonable. The ID used in Figure 2 is the 32-bit hexadecimal value `12345678`. cite?

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 might be able to trigger an infinite loop of executing the ID-check opcode bytes `75...E1`, and might even (if one is not careful about the use of registers) be able to circumvent CFI. forget?

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

Section 6 describes machine-code sequences that build on the two alternatives in this figure.

### 2.3 Assumptions

If CFI enforcement is to be implemented by inserting, at each source and destination, particular machine-code instruction sequences that implement ID-checks and IDs, respectively, it is critical that three assumptions hold. These three assumptions, implicit above, 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, for software of reasonable size, by making the space of IDs large enough (say, 32-bit) and by choosing IDs so that they do not conflict with the code.

**NWC** Non-Writable Code: An attacker must not be able to modify code memory at runtime. Otherwise, the attacker could trivially circumvent CFI, for example by overwriting ID-checks. 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 to execute data as if it were code. Otherwise, an attacker could cause the execution of data that is labelled 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. cite?

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 [8] and other mitigation mechanisms [7, 35]. Since a lucky, persistent, or knowledgeable attacker will still succeed [28], we do not consider this

variant attractive, and we do not discuss it further. Instead, Section 4 shows how, for systems missing NXD hardware support, CFI enforcement can be integrated with a 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. Even so, we are currently working on expanding inlined CFI enforcement to handle runtime code generation.

The implementation of IDs and ID-checks is applicable even to multi-threaded programs, since external threads cannot affect the registers used in ID-checks. However, this property holds only if the program in question does not employ user-level threads, and if the program cannot make system calls that arbitrarily change system state; without this assumption, which we call **SYS** from now on, not only can one thread modify the registers of other threads, but UNQ, NWC, and NXD are also trivially violated. Although SYS is made by most security enforcement mechanisms, it is usually left unstated. In practice, SYS can be difficult to satisfy without support from the operating system. As discussed further in Section 4, CFI makes it easier to satisfy SYS, by directly identifying system call sites and facilitating restrictions on their arguments.

### 2.4 The 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 from the literature (e.g., [31]), for instance at compile time. The x86 implementation of Section 6 works with binaries (rather than source code) and refines its control-flow analysis by using type signature information. cite?

After CFI instrumentation (perhaps at install-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. cite?

Finally (for example, when a program is loaded into memory and assembled from components and libraries), a CFI verification phase can 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.

## 3 A Formal Analysis of Inlined CFI Enforcement

In this section we present a formal treatment of our basic protection mechanism. This formal treatment includes a precise semantics for programs and definitions for program

instrumentation. It also includes precise guarantees about the executions of instrumented programs.

In our experience, a formal approach is useful not only for elucidating assumptions and guarantees, but also as a guide in the design and development of our techniques. Indeed, in the course of our work, we rejected several alternatives that relied on unclear assumptions or that offered guarantees only in hard-to-define circumstances.

For the sake of simplicity we work with a small set of machine instructions which enables us to address CFI but excludes some features found in actual architectures. We have formally analyzed other aspects of our machinery. We omit details because (in our opinion) some of the formal elaborations yield diminishing returns, and for the sake of brevity.

### 3.1 Programs and Machine Model

The programs and the machine model that we treat in this section are fairly mundane, and typical of formal studies. We write programs in terms of a small set of instructions similar to those of the previous section. Essentially, our language is that of Hamid et al. [12] plus a *nop* instruction in which an immediate value can be embedded. The set of instructions is:

$$\begin{aligned} \mathit{Instr} \;=\; \{\, & \mathit{nop}\ w, \\ & \mathit{add}\ r_d, r_s, r_t, \\ & \mathit{addi}\ r_d, r_s, w, \\ & \mathit{movi}\ r_d, w, \\ & \mathit{bgt}\ r_s, r_t, w, \\ & \mathit{jd}\ w, \\ & \mathit{jmp}\ r_s, \\ & \mathit{ld}\ r_d, r_s(w), \\ & \mathit{st}\ r_d(w), r_s, \\ & \mathit{illegal}\,\} \end{aligned}$$

where $w$ is a word and $r_s$, $r_t$, and $r_d$ are registers. Thus, instructions may contain words. Like Hamid et al., we omit the routine details of instruction storage and decoding. We assume a function $Dc : \mathit{Word} \to \mathit{Instr}$ that decodes words into instructions. We define words, register files, and states as follows:

$$\begin{aligned} w, \mathit{pc} \in \mathit{Word} &= \{0, 1, ...\} \\ M \in \mathit{Mem} &= \mathit{Word} \to \mathit{Word} \\ R \in \mathit{Regfile} &= \mathit{Regnum} \to \mathit{Word} \\ S \in \mathit{State} &= \mathit{Mem} \times \mathit{Regfile} \times \mathit{Word} \end{aligned}$$

The formal semantics of instructions is given in terms of state transitions in Section 3.2.

When $S$ is a state, we may write $S.M$, $S.R$, and $S.pc$ for the *Mem* component, the *Regfile* component, and the *pc* in $S$, respectively. We further distinguish between code memory

($M_c$) and data memory ($M_d$), so we split the memory into two functions with disjoint domains. We assume that a statically defined program containing $n > 0$ instructions always occupies memory locations 0 to $n - 1$, with the first instruction of the program located at address 0. When we split a general memory $M$ into $M_c$ and $M_d$, we write $M = M_c|M_d$, provided $M_c$ contains $n > 0$ instructions and the following constraints are met:

$$\begin{array}{l} \mathrm{dom}(M_c) = \{0..(n-1)\} \\ \mathrm{dom}(M_d) = \mathrm{dom}(M) - \mathrm{dom}(M_c) \\ M_c(a) = M(a) \quad \text{for all } a \in \mathrm{dom}(M_c) \\ M_d(a) = M(a) \quad \text{for all } a \in \mathrm{dom}(M_d) \end{array}$$

We consider only states whose memory is partitioned in this way. We write $S.M_c$ to indicate the code memory of state $S$, and $S.M_d$ for the data memory.

Similarly, we split the register file into distinguished and general registers. When we split a general register file $R$ into $R_{0-2}$ and $R_{3-31}$, we write $R = R_{0-2}|R_{3-31}$ provided the following constraints are met:

$$\begin{array}{l} \mathrm{dom}(R_{0-2}) = \{r_0, r_1, r_2\} \\ \mathrm{dom}(R_{3-31}) = \{r_3..r_{31}\} \\ R_{0-2}(r) = R(r) \quad \text{for all } r \in \mathrm{dom}(R_{0-2}) \\ R_{3-31}(r) = R(r) \quad \text{for all } r \in \mathrm{dom}(R_{3-31}) \end{array}$$

The distinguished registers $r_0$, $r_1$, and $r_2$ will be assumed to be used only in the code inserted at instrumentation time.

### 3.2 Semantics of Programs under Attack

In this section we give a semantics for instructions. The main definitions are in Figure 3.

- The relation $\rightarrow_n$ models normal small steps of execution, that is, those steps that may occur in the absence of an attacker. This relation is deliberate incomplete: many states are "stuck", including those where $Dc(M_c(pc)) = \mathit{illegal}$.

- The relation $\rightarrow_a$ models attack steps. In such a step, an attacker may unconditionally and arbitrarily perturb data memory and/or undistinguished registers.

  For example, the attacker may modify a part of memory to contain a bit pattern that appears elsewhere in memory. Thus, intuitively, the attacker can read all of memory.

- The relation $\rightarrow$ is the union of $\rightarrow_n$ and $\rightarrow_a$. Thus, this relation represents a computation step in general, either a normal state transition or one caused by an attacker.

### 3.3 Discussion of Assumptions

The definitions above embody several assumptions:

- The definition of $\rightarrow_n$ implies NXD. Similarly, the definitions of $\rightarrow_n$ and $\rightarrow_a$ imply NWC. (See Section 5 for an alternative.)

- The definition of $\rightarrow_a$ implies that the attacker cannot modify the distinguished registers $r_0$, $r_1$, and $r_2$. This assumption is conservative, and can be seen as a formal counterpart to SYS: without SYS, an attacker could modify any register at any time. Our proofs require only a weaker assumption, namely that the attacker cannot modify the distinguished registers during the execution of code inserted at instrumentation time.

- The machine model and the definition of $\rightarrow_n$ exclude the possibility that a jump would land in the middle of an instruction. CFI can prevent jumps into the middle of an instruction. Indeed, our implementation is designed to achieve this property. For simplicity, we do not address it in the formal analysis.

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 external radiation might affect instruction semantics in arbitrary ways.

### 3.4 The CFG

The instrumentation of a program relies on a CFG for the program. The nodes of the CFG are words that represent program addresses. Given a graph $G$ for $M_c$, and $w \in \mathrm{dom}(M_c)$, we let $\mathrm{succ}(w)$ be the set of words $w' \in \mathrm{dom}(M_c)$ such that $G$ has an edge from $w$ to $w'$. We typically write $\mathrm{succ}(M_c, w)$, making $M_c$ explicit but leaving $G$ implicit when it is clear from context (particularly because $G$ may often be determined from $M_c$).

We need not make assumptions on how this graph is obtained, or how it matches the executions of the program before instrumentation. However, we do require:

1. If $Dc(M_c(w_0)) = \mathit{nop}\ w$, or $\mathit{add}\ r_d, r_s, r_t$, or $\mathit{addi}\ r_d, r_s, w$, or $\mathit{movi}\ r_d, w$, or $\mathit{ld}\ r_d, r_s(w)$, or $\mathit{st}\ r_d(w), r_s$, then $\mathrm{succ}(w_0) = \{w_0 + 1\} \cap \mathrm{dom}(M_c)$.

2. If $Dc(M_c(w_0)) = \mathit{bgt}\ r_s, r_t, w$ then $\mathrm{succ}(w_0) = \{w_0 + 1, w\} \cap \mathrm{dom}(M_c)$.

3. If $Dc(M_c(w_0)) = \mathit{jd}\ w$ then $\mathrm{succ}(w_0) = \{w\} \cap \mathrm{dom}(M_c)$.

4. If $Dc(M_c(w_0)) = \mathit{jmp}\ r_s$ then $\mathrm{succ}(w) \neq \emptyset$.

5. $Dc(M_c(w_0)) = \mathit{illegal}$ then $\mathrm{succ}(w_0) = \emptyset$.

6. If $w_0, w_1 \in \mathrm{dom}(M_c)$, then $\mathrm{succ}(w_0) \cap \mathrm{succ}(w_1) = \emptyset$ or $\mathrm{succ}(w_0) = \mathrm{succ}(w_1)$.

These properties hold for many graphs that arise from code analysis. Only the last one (6) is non-trivial. When it does not hold, we can easily make it hold—we have a construction that achieves this property by duplicating nodes.

We say that $w'$ is a destination if there exists $w$ such that $Dc(M_c(w))$ is a jump instruction ($\mathit{jmp}\ r_s$) and $w' \in \mathrm{succ}(M_c, w)$. Because of property 6, we can put destinations into equivalence classes. We give each equivalence class an identifier, called an ID. We represent

| If $Dc(M_c(pc))=$ | then $(M_c \mid M_d, R, pc) \rightarrow_n$ |
|---|---|
| $nop\ w$ | $(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$ |
| $add\ r_d, r_s, r_t$ | $(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$ |
| $addi\ r_d, r_s, w$ | $(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$ |
| $movi\ r_d, w$ | $(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$ |
| $bgt\ r_s, r_t, w$ | $(M_c \mid M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$<br>$(M_c \mid M_d, R, pc+1)$, when $R(r_s) \leq R(r_t) \wedge pc+1 \in \mathrm{dom}(M_c)$ |
| $jd\ w$ | $(M_c \mid M_d, R, w)$, when $w \in \mathrm{dom}(M_c)$ |
| $jmp\ r_s$ | $(M_c \mid M_d, R, R(r_s))$, when $R(r_s) \in \mathrm{dom}(M_c)$ |
| $ld\ r_d, r_s(w)$ | $(M_c \mid M_d, R\{r_d \mapsto M(R(r_s)+w)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$ |
| $st\ r_d(w), r_s$ | $(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$ |

$$\frac{}{(M_c \mid M_d, R_{0-2} \mid R_{3-31}, pc) \rightarrow_a (M_c \mid M_d{}', R_{0-2} \mid R_{3-31}{}', pc)} \quad \text{(A-Step)}$$

$$\frac{S \rightarrow_n S'}{S \rightarrow S'} \quad \text{(G-Step1)}$$

$$\frac{S \rightarrow_a S'}{S \rightarrow S'} \quad \text{(G-Step2)}$$

Figure 3: Normal steps, attacker steps, and general steps

these IDs by words. For a *jmp* instructions at address $w$ in $M_c$, we let $\mathrm{dst}(M_c, w)$ be the ID of all successors of $w$; thus, $\mathrm{dst}(M_c, w)$ is the ID of any element of $\mathrm{succ}(M_c, w)$.

### 3.5 Instrumentation

For the present purposes, instrumentation consists in ensuring that a code memory $M_c$ satisfies the following properties, either by checking them or (more probably) by rewriting code so that they hold:

1. The final instruction in $M_c$ is *illegal*.

2. If $w_0 \in \mathrm{dom}(M_c)$ is a destination, then the instruction at $w_0$ is *nop* $w$, where $w$ is $w_0$'s ID.

3. If $w_0 \in \mathrm{dom}(M_c)$ holds a *jmp* instruction, then this instruction is *jmp* $r_0$ and it is in the context of the following code:

   $$\begin{array}{l} addi\ r_0, r_s, 0 \\ ld\ r_1, r_0(0) \\ movi\ r_2, IMM \\ bgt\ r_1, r_2, HALT \\ bgt\ r_2, r_1, HALT \\ jmp\ r_0 \end{array}$$

   where $r_s$ is some register, $HALT$ is the address of the *illegal* instruction of (1) and $IMM$ is the word $w$ such that $Dc(w) = nop\ \mathrm{dst}(M_c, w_0)$. This code compares the dynamic target of a jump, which is initially in register $r_s$, to the *nop* instruction that is expected to be the target statically. When the comparison succeeds, the jump proceeds. When it fails, the program halts.

4. *bgt* and *jd* instruction do not target *jmp* instructions or any of the preceding instructions $ld\ r_1, r_0(0)$, $movi\ r_2, IMM$, $bgt\ r_1, r_2, HALT$, and $bgt\ r_2, r_1, HALT$ described in (3). They may target instructions $addi\ r_0, r_s, 0$.

   (Note that (2) removes the possibility that a *jmp* instruction can jump to another *jmp* instruction or to any of the preceding instructions considered here.)

The predicate $I(M_c)$ indicates $M_c$ satisfies these properties.

### 3.6 A Theorem about CFI

With these definitions, we can obtain formal results about our instrumentation method. Here we present only a simple but fundamental result that expresses integrity of control flow. The following theorem states 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. Thus, despite attack steps, the program counter always follows the CFG.

**Theorem 1**
*Let $S_0$ be a state $(M_c|M_d, R, pc)$ such that $I(M_c)$ and $pc = 0$, and let $S_1, \ldots, S_n$ be states such that $S_0 \to S_1 \to ... \to S_n$. Then, for all $i \in 0..(n-1)$, either*

$$S_i \to_a S_{i+1} \text{ and } S_{i+1}.pc = S_i.pc$$

*or*

$$S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)$$

Appendix A contains additional technical developments and proofs.

Although this theorem is fairly easy to state, it has strong consequences. In particular, it implies that the attacker cannot cause the execution of code that would appear unreachable according to the CFG. For example, if the system "execute" command should not be reachable in a certain code memory, then it will not run. chat?

# 4 CFI Applications and Refinements

CFI ensures that runtime execution proceeds along a static CFG—guaranteeing, for instance, that the execution of a typical function always starts at the top and proceeds from beginning to end. Thereby, CFI can increase the reliability of any CFG-based technique (e.g., strengthening the techniques against buffer overflows and intrusions in [15] and [31], respectively). However, this is not the only implication of CFI.

## 4.1 CFI as a Foundation for IRMs

CFI greatly facilitates Inlined Reference Monitors, or IRMs, which enforce security policies by inserting into target programs the code for validity checks and also any additional security-relevant state needed [17]. IRMs require that the target program can neither circumvent the inserted validity checks nor subvert the added security state. By constraining the CFG enforced by CFI, the first of these requirements is easily satisfied.

One application of CFI could be the IRM enforcement of security policies that restrict a program's use of the operating system (for instance, preventing files with some filenames from being written). Such policies are often necessary—even for CFI, as noted by SYS in Section 2.4—and existing mechanisms modify operating systems in ways that are best avoided [26]. With CFI, it is easy to enumerate those points in a program where system calls can be made at runtime. At each such point, the code for an IRM validity check can be inserted, and CFI can ensure the check cannot be circumvented. cite!

Even so, under the powerful attack model used in this paper, CFI alone cannot guarantee that IRM security state is not subverted. Any data memory is modifiable by the attacker, including the filenames used in system calls that read and write files. By generalizing SFI, however, CFI can be extended to support isolated data memory regions—and, at the same time, the NWC and NXD assumptions of CFI can be eliminated.

## 4.2 SMAC: SFI Revisited

Software Fault Isolation, or SFI, is one particular type of IRM, designed to emulate traditional memory protection. In SFI, code is inserted at each machine-code instruction that

```
call  eax                              ret
```

with SMAC eliminating the NXD assumption, can become (a):

```
and  eax, 00FFFFFFh                    mov  ecx, [esp]
cmp  ds:[eax+4], ID                    and  ecx, 00FFFFFFh
jne  error_label                       cmp  ds:[ecx+4], ID
call eax                               jne  error_label
prefetchnta ID                         add  esp, 4h
                                       jmp  ecx
```

with a SMAC implementation of a shadow call stack, can become (b):

```
    sub  fs:[338h], 4h                 mov  ecx, fs:[338h]
    mov  ecx, fs:[338h]                mov  ecx, [ecx]
    mov  [ecx], LRET                   add  fs:[338h], 4h
    cmp  ds:[eax+4], ID                add  esp, 4h
    jne  error_label                   jmp  ecx
    call eax
LRET: ...
```

Figure 4: SMAC instrumentation, for two different purposes, of x86 `call` and `ret`.

accesses memory. to ensure the target memory address lies within a certain range [33, 19]. Building on CFI not only makes SFI-inserted code non-circumventable but can also greatly reduce SFI enforcement overhead. This optimization is enabled by a guaranteed CFG, since this removes the need to repeatedly check memory addresses such as local variables.

Building on CFI, Software Memory Access Control, or SMAC, extends SFI to allow different inserted checks at different instructions in the program. Thereby, SMAC can enforce policies other than those of traditional memory protection. In particular, SMAC can create *isolated data memory* that is accessible from only certain program code, for instance, from a library function or even individual instructions. Thus, SMAC can be used to implement IRM security state 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.

SMAC can also be used to eliminate CFI assumptions: either NWC, by disallowing writes to certain memory addresses, or NXD, by preventing control flow outside those addresses. Figure 4a shows SMAC instrumentation that can guarantee only code is executed. In the figure, the top eight bits are masked off the destination addresses of x86 function calls and returns, using a single `and` instruction. SMAC checks may be as simple as this single instruction, since they are greatly simplified by the CFI guarantee of a runtime CFG.

### 4.3 Enforcement of More Precise CFI Policies

Preferably, CFI enforcement should be as precise as possible. However, when based on IDs and ID-checks, CFI can be more permissive than necessary because the ID of a destination must be used by the ID-check of all sources for that destination. As a result, ID-based CFI enforcement cannot enforce exactly those CFGs where sources go to overlapping but move?

unequal destination sets: because they overlap, the sets will be assigned the same ID and CFI will allow control flow from the sources to the union of the destination sets.[2]

One general strategy for increasing the precision of the CFG is code duplication: CFG nodes (such as basic blocks or functions) with a high degree of computed control-flow edges can be split into multiple nodes. This strategy might, for instance, be applied to programs compiled from languages with subtyping—since, whenever a method $m$ is invoked on a subtype $T$, ID-based CFI enforcement would allow control flow to all $m$ implementations, including those in supertypes $S$.

However, in the case of subtyping, as well as in many other cases, changing the CFI instrumentation may be a better option for increasing CFG precision. For example, if implementations of subtype methods are laid out in ascending order in code memory, then adding a single comparison to the ID-checks may suffice to prohibit invocation of supertype methods. Alternatively, more than one ID could be added to certain destinations, or ID-checks could sometimes compare against only certain bits of the destination ID.

Because CFI enforces a static CFG, it cannot ensure that functions return to the actual call-site used to invoke them—and using runtime information to guarantee this is another (perhaps more important) way to increase the precision of CFI enforcement. For example, without this refinement, in Figure 1 of Section 2.1 a powerful attacker could prevent the middle part of `sort2` from ever being executed.

A *shadow call stack* is one mechanism that can guarantee correct destinations of returns [11, 25, 20]. Using SMAC isolated data memory, CFI enforcement can be adapted to implement a shadow call stack that cannot be corrupted by an attacker. This protection is possible because SMAC can ensure only function call and return instructions can modify the shadow call stack—and only by correctly pushing and popping the correct values.

Figure 4b shows a portion of the SMAC instrumentation of a protected shadow call stack. In the figure, `fs:[338h]` points to the top of the shadow call stack, which is assumed to reside in isolated data memory outside the range of the program's write instructions. Note that, in Figure 4b, inlined CFI enforcement need not perform an ID-check on function return: the protected shadow call stack implies that the destination is correct.

Efficient implementation of protected shadow call stacks probably requires some support from the operating system—since, in modern systems, more than one stack may be used for different threads, and their size may grown dynamically. For instance, multiple instructions may be required to detect stack underflow and overflow, whereas an operating system could make use of hardware-supported guard pages that come with at no performance penalty. The SMAC instrumentation in Figure 4b already assumes a thread-local `fs` segment (such as that of Microsoft Windows), which is context-switched by operating system. Section 6 gives some results on SMAC performance that assume certain operating system and hardware support of this type.

[2] This permissiveness of ID-based CFI enforcement does not affect function return—the most common source of computed control flow—but may, in practice, be an issue for programs that make heavy use of function pointers.

| If $Dc(M(pc))=$ | then $(M, R, pc) \rightarrow_n$ |
|---|---|
| $nop\ \ w$ | $(M, R, pc+1)$ |
| $add\ \ r_d, r_s, r_t$ | $(M, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc+1)$ |
| $addi\ \ r_d, r_s, w$ | $(M, R\{r_d \mapsto R(r_s) + w\}, pc+1)$ |
| $movi\ \ r_d, w$ | $(M, R\{r_d \mapsto w\}, pc+1)$ |
| $bgt\ \ r_s, r_t, w$ | $(M, R, w)$, when $R(r_s) > R(r_t)$<br>$(M, R, pc+1)$, when $R(r_s) \leq R(r_t)$ |
| $jd\ \ w$ | $(M, R, w)$ |
| $jmp\ \ r_s$ | $(M, R, R(r_s))$ |
| $ld\ \ r_d, r_s(w)$ | $(M, R\{r_d \mapsto M(R(r_s) + w)\}, pc+1)$ |
| $st\ \ r_d(w), r_s$ | $(M\{R(r_d) + w \mapsto R(r_s)\}, R, pc+1)$ |

Figure 5: Normal steps (without NXD and NWC)

# 5 A Formal Analysis of Inlined CFI Enforcement, with SMAC

This section presents a basic formal treatment of SMAC, and continues the formal analysis of inlined CFI enforcement with SMAC.

## 5.1 An Alternative Program Semantics

With SMAC, the assumptions NXD and NWC are no longer needed. We can therefore relax the program semantics of Section 3.2. The resulting definition of normal execution steps is in Figure 5. The definitions of attack steps and general steps remain those of Figure 3. In particular, we still require that an attack step cannot directly alter code memory, the distinguished registers, or the program counter. Without such restrictions, an attacker could obviously create new code or jump arbitrarily to existing code. Those restrictions do not apply to the normal steps. Of course, normal steps could conceivably execute as a result of some action by the attacker, and we aim to limit the harm that they can cause.

## 5.2 Instrumentation, with SMAC

We assume that the minimum and maximum addresses of code and data memory are known at instrumentation time, and let $\min(M)$ and $\max(M)$ respectively return the minimum and maximum addresses in the domain of memory $M$. SMAC-based instrumentation consists in ensuring that a code memory $M_c$ satisfies the following properties:

1. The final instruction in $M_c$ is *illegal*.

2. If $w_0 \in \mathrm{dom}(M_c)$ is a destination, then the instruction at $w_0$ is *nop* $w$, where $w$ is $w_0$'s ID.

3. If $w_0 \in \mathrm{dom}(M_c)$ holds a *st* instruction, then this instruction is *st* $r_0(0), r_s$ and it is in the context of the following code:

$$\begin{array}{l} addi\ r_0, r_d, w \\ movi\ r_1, \max(M_d) \\ movi\ r_2, \min(M_d) \\ bgt\ r_0, r_1, HALT \\ bgt\ r_2, r_0, HALT \\ st\ \ r_0(0), r_s \end{array}$$

   where $r_d$ is some register and *HALT* is the address of the *illegal* instruction of (1).

4. If $w_0 \in \mathrm{dom}(M_c)$ holds a *jmp* instruction, then this instruction is *jmp* $r_0$ and it is in the context of the following code:

$$\begin{array}{l} addi\ r_0, r_s, 0 \\ movi\ r_1, \max(M_c) \\ movi\ r_2, \min(M_c) \\ bgt\ r_0, r_1, HALT \\ bgt\ r_2, r_0, HALT \\ ld\ r_1, r_0(0) \\ movi\ r_2, IMM \\ addi\ r_2, r_2, 1 \\ bgt\ r_1, r_2, HALT \\ bgt\ r_2, r_1, HALT \\ jmp\ r_0 \end{array}$$

   where $r_s$ is some register, *HALT* is the address of the *illegal* instruction of (1) and *IMM* is the word $w$ such that $Dc(w) = nop\ \mathrm{dst}(M_c, w_0)$.

5. *bgt* and *jd* instruction do not target *st* instructions or any of the preceding instructions listed in (3), or *jmp* instructions or any of the preceding instructions listed in (4), except possibly the first of these instructions, namely *addi* $r_0, r_d, w$ and *addi* $r_0, r_s, 0$, respectively.

The predicate $I_s(M_c)$ indicates $M_c$ satisfies these properties.

### 5.3 A Theorem about CFI with SMAC

With the relaxed semantics and the SMAC instrumentation, we obtain a direct counterpart to Theorem 1.

**Theorem 2**
*Let $S_0$ be a state $(M_c|M_d, R, pc)$ such that $I_s(M_c)$ and $pc = 0$, and let $S_1, \ldots, S_n$ be states such that $S_0 \to S_1 \to ... \to S_n$. Then, for all $i \in 0..(n-1)$, either*

$$S_i \to_a S_{i+1} \text{ and } S_{i+1}.pc = S_i.pc$$

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

are CFI instrumented using `prefetchnta` destination IDs, to become

```
8B 44 B3 08  mov  eax, [ebx+8]
3E 81 78 04  cmp  ds:[eax+4],
78 56 34 12       12345678h
75 13        jne  error_label
FF E0        call eax
3E 0F 18 05  prefetchnta
DD CC BB AA     ds:[AABBCCDDh]
```

```
8B 0C 24     mov  ecx, [esp]
83 C4 14     add  esp, 14h
3E 81 79 04  cmp  ds:[ecx+4],
78 56 34 12       AABBCCDDh
75 13        jne  error_label
FF E1        jmp  ecx
```

Figure 6: The x86 CFI instrumentation of `call` and `ret` used in the experiments.

*or*

$$S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)$$

See Appendix B for the corresponding proofs.

# 6 Practical CFI Implementation

**Draft rough outline** IRM Implementation using Vulcan.

Can create the "NOP ID" in many ways–e.g., by adding and subtracting the same byte constant from the AL register (assuming it is free, as it will be at start of most functions), or even by assigning a 32-bit constant to EAX. The prefetch instruction has some advantages, most notably that it allows 32-bit alignment to be preserved, both for start of the function and also for accesses to the constant ID.

Because of the "infinite-loop trick", must make sure that the return IDs used in calls and returns are disjoint—otherwise a call ID-check could allow a jump into the middle of a return ID-check, without any restrictions on the ECX register. Same problem, with the EAX register, in the opposite direction.

- Change CALL/RET/ENTER/LEAVE to jmp R

Should be fast (make good use of caches): Checks and IDs same locality as code Static pressure on unified caches and top-level iCache Dynamic pressure on top-level dTLB and dCache

Caveats with system calls, dynamic libraries, etc. (Point out same caveat also applies to most previous work.)

How to coarsen CFG to maintain backwards compatibility etc.

Performance benchmarks.

Implementing NXD in OS using TLB trcks etc.

Ways to improve performance that add new security vulnerabilities (race conditions and/or information disclosure attacks).

This enforcement style avoids extra data structures, since identifying bit patterns are embedded within the inserted machine-code, e.g., at start of functions, as an immediate

constant in an effective-`nop` instruction. Enforcing CFI in this manner is also efficient, in large part because the inserted checks and bit-patterns have the same locality properties as executing code, and therefore avoid the penalties of high memory latencies. CFI implementation is discussed further in Sections 2 and 6

It is important that, for security purposes, there is a great deal of flexibility in how the fundamental CFI guarantees are implemented-with (almost) a different implementation for each different point spectrum of precision, security, efficiency, and compatibility with existing machine-code programs. At one extreme, the "security" of a CFI implementation can be proven given a machine-code semantics, and at other extremes the CFI implementation may allow many race-condition and/or information-disclosure-based attacks, yet still be quite secure in practice. The granularity of the CFG enforced by the CFI mechanism is one important dimension by which exactness can be traded against other factors-for example, by enforcing a simpler (and thus coarser-grained) CFG, security may be lessened but compatibility with arbitrary pre-existing programs (such as from handwritten machine code) can be increased.

### 6.1 Implementing CFI on Windows/x86

**Draft rough outline**

### 6.2 CFI Performance Measurements

**Draft rough outline**

## 7 Related Work

As mentioned in the introduction, there is a variety of related work on vulnerability mitigation techniques, including that in [3, 4, 5, 8, 6, 7, 9, 11, 16, 21, 20, 14, 23, 25, 27, 29, 30, 35]. Also related is the work on security policy enforcement in [19, 18, 17, 26], the work on CFG-based anomaly detection in [10, 15, 31, 32], and the attack literature, such as [1, 2, 28, 22, 24, 34]. We focus, in this section, on the three mitigation techniques most closely related to CFI.

PointGuard [7] stores code addresses in an encrypted form in data memory. Here, the idea is that, even if attackers can change data memory, they cannot ensure that control flows to an specific address: for this, they must know the encryption key. (This underlying idea is also used in [4, 8, 23, 35, 30].) Adopting PointGuard can be difficult, since all program points that use code addresses or depend upon their encoding must be correctly identified and modified. More worryingly, to a lucky, persistent, or knowledgeable attacker, PointGuard and similar schemes do not restrict control flow in any way [28, 30]. In practice, PointGuard is only applicable to hard-to-define aspects of a target program's semantics determine whether PointGuard can be applied or offer security benefits to that program.

In

In practice,

whether PointGuard can offers security the target program semantics Whether these problems apply

Program Shepherding

a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms. Unlike intrusion detection systems based on control flow, CFI is not subject to mimicry attacks.

Program shepherding proposed adding CFG table to binaries: in CFI this table is embedded implicitly within the code.

CFI would have prevented most well-known epidemic worms and viruses, including XXX and YYY, as they were written and deployed. However, it is unclear how XXX and YYY would have been constructed if CFI was widely adopted.

Note that our work actually prevents most jump-to-`libc` attacks, since it only allows such library calls from places where they already exist in the binary—forcing the attacker to find a vulnerability that is closely coupled with use of a dangerous library routine. One way to prevent this is to enforce a more precise CFG, such as by forcing each function to return to its runtime call site. This can also be mitigated somewhat by using randomization etc.

Constraining software control flow for security purposes has been explored before, most notably in *Program Shepherding* [13]. This previous work uses an enforcement mechanism based on runtime machine-code translation, and coalesces control-flow guarantees with several other security policies. The current paper contributes, in part, by isolating and clearly defining the CFI policy, by formally proving CFI properties against powerful attackers, and by presenting inlined CFI enforcement. Inlined CFI is practical and efficient, and can also be trustworthy, because its implementation is simple, based on formal reasoning, and doesn't need to rely on complex analysis or mechanisms. cite TR?

# 8 Conclusion

For a long time, high-level programming languagess have implied strict restrictions on software execution, in particular its control flow. Despite this, operating systems and hardware have continued to assume that all could be reached from potentially any machine-code instruction—at most, making a coarse distinction between code and data. CFI, and its inlined enforcement, fundamentally changes this situation, thereby enabling the use of many important software technologies, as well as providing strong security guarantees.

Our CFI definition and strategy for implementation using program rewriting is superior to all previous approaches because: (need to fine tune this text and/or move)

- CFI is a simple enforcement mechanism whose guarantees can be easily stated, yet thwarts many attacks from powerful attackers.
- These CFI guarantees are strong: they are not probabilistic or subject to information disclosure attacks—as long as we execute processors that support NXD (or if we use CFI to do SMAC).
- CFI properties can be formally proven/verified given a formal model of machine-code semantics.
- CFI enforcement by software instrumentation is more efficient than techniques with similar guarantees on control flow.

- (Also, hardware CFI enforcement can be trivially added to processors and would have zero performance/power/die cost.)
- By choosing CFG identifier bit patterns, CFI can enforce any level of backwards compatibility, even with handwritten machine code.
- CFI instrumentation creates self-describing software binaries, whose runtime control-flow can be derived from their content.
- Having a guaranteed runtime CFG facilitates all program analysis and processing—in particular allowing many optimizations.
- The CFI trustworthiness can be isolated using PCC-style *verification*, so CFI contributes no complexity to the TCB.
- For statically linked programs, CFI does not require any runtime mechanism, except for minor OS restrictions—and these are also required for all other mechanisms, whether they admit it or not.
- Finally, CFI supports building of additional security, such as SFI, dynamic library loading, and in general any IRMs.

**Acknowledgments** Martín Abadi and Jay Ligatti participated in this work while at Microsoft Research, Silicon Valley. Discussions with Greg Morrisett were helpful to this paper's development and improved its exposition.

## References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[2] Anonymous. Bypassing PaX ASLR protection. *Phrack*, 11(59), Jul 2002.

[3] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *13th USENIX Security Symposium*, San Diego, CA, August 2003.

[4] S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium*, San Diego, CA, August 2003.

[6] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. of the 10th Usenix Security Symposium*, Aug 2001.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.

[8] 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 *Proc. of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998.

[9] J.R. Crandall and F.T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 2004: 37th Annual International Symposium on Microarchitecture*, Portland, OR, December 2004.

[10] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2003.

[11] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, Washington, DC, August 2001.

[12] 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, Jan 2002.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th Usenix Security Symposium*, Aug 2002.

[14] D. Kirovski and M.Drinic. POPI — a novel platform for intrusion prevention. In *MICRO 2004: 37th Annual International Symposium on Microarchitecture*, Portland, OR, December 2004.

[15] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, Washington, DC, August 2001.

[16] E. Larson and T. Austin. High coverage detection of input-related security faults. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[17] Úlfar Erlingsson. *The Inlined Reference Monitor Approach To Security Policy Enforcement*. PhD thesis, Cornell University, 2004.

[18] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[19] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[20] D. Nebenzahl and A. Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. In *19th IFIP International Information Security Conference*, Toulouse, France, August 2004.

[21] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[22] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001.

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

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

[25] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proc. of the 2003 Usenix Annual Technical Conference*, Jun 2003.

[26] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[27] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[28] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security*, 2004.

[29] G.E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS 2004: 11th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96. ACM Press, 2004.

[30] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *MICRO 2004: 37th Annual International Symposium on Microarchitecture*, Portland, OR, December 2004.

[31] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, California, 2001.

[32] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *ACM Conference on Computer and Communications Security*, 2002.

[33] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[34] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. of the 10th Network and Distributed System Security Symposium*, Feb 2003.

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

# Appendix

# A Further Formal Material on CFI

## A.1 What Instrumentation Implies

Instrumentation immediately leads to several static constraints on code memory. The definitions of the constraints and the analysis in the next subsection use the predicates defined below.

- $\mathrm{MIC}(M_c, w) \iff \exists k \in \{0..4\} \exists r_s. Dc(M_c(w+k)) = jmp\ r_s$
  ($w$ is in the middle of instrumented code before a *jmp* instruction)
- $\mathrm{TB}(S_1, S_2) \iff \left( \begin{array}{l} S_1 \rightarrow_n S_2 \ \wedge \\ \quad \exists r_s. Dc(S_1.M_c(S_1.pc)) = jmp\ r_s \ \vee \\ \quad \exists w. Dc(S_1.M_c(S_1.pc)) = jd\ w \ \vee \\ \quad\quad \exists r_s \exists r_t \exists w. Dc(S_1.M_c(S_1.pc)) = bgt\ r_s, r_t, w \ \wedge \\ \quad\quad S_2.pc \neq S_1.pc + 1 \end{array} \right)$
  ($S_1$ takes a branch to get to $S_2$)

The static constraints on code memory induced by the instrumentation algorithm follow.

[I-Jmp] $\forall M_c \ \forall a \in \mathrm{dom}(M_c) \ \forall r_s :$

$$Dc(M_c(a)) = jmp\ r_s \Rightarrow \left( \begin{array}{l} \exists r'_s : Dc(M_c(a-5)) = addi\ r_0, r'_s, 0 \ \wedge \\ Dc(M_c(a-4)) = ld\ r_1, r_0(0) \ \wedge \\ \exists w_1\ \exists w_2\ \forall a' \in \mathrm{dom}(M_c) : \\ \qquad Dc(M_c(a-3)) = movi\ r_2, w_1 \ \wedge \\ \qquad Dc(w_1) = nop\ w_2 \ \wedge \\ \qquad Dc(M_c(a')) = nop\ w_2 \Rightarrow a' \in \mathrm{succ}(M_c, a) \ \wedge \\ \exists w_3 : Dc(M_c(a-2)) = bgt\ r_1, r_2, w_3 \ \wedge \\ \qquad Dc(M_c(a-1)) = bgt\ r_2, r_1, w_3 \ \wedge \\ \qquad Dc(M_c(w_3)) = illegal \ \wedge \\ r_s = r_0 \end{array} \right)$$

[I-Jd] $\forall M_c \forall a \in \mathrm{dom}(M_c) \forall w.$

$$Dc(M_c(a)) = jd\ w \Rightarrow \left( \begin{array}{l} w \in \mathrm{succ}(M_c, a) \ \wedge \\ \neg \mathrm{MIC}(M_c, w) \end{array} \right)$$

[I-Bgt] $\forall M_c \forall a \in \mathrm{dom}(M_c) \forall r_s \forall r_t \forall w.$

$$Dc(M_c(a)) = bgt\ r_s, r_t, w \Rightarrow \left( \begin{array}{l} w \in \mathrm{succ}(M_c, a) \ \wedge \\ \neg \mathrm{MIC}(M_c, w) \end{array} \right)$$

[I-Fall] $\forall M_c \forall a \in \mathrm{dom}(M_c).$

$$\left( \begin{array}{l} Dc(M_c(a)) \neq illegal \ \wedge \\ \forall r_s. Dc(M_c(a)) \neq jmp\ r_s \ \wedge \\ \forall w. Dc(M_c(a)) \neq jd\ w \end{array} \right) \Rightarrow a+1 \in \mathrm{succ}(M_c, a)$$

## A.2 Lemmas on CFI Enforcement

A start state is any state of the form $(M, R, 0)$. Throughout, we let $S_0$ range over start states.

**Lemma 3**
$\forall n \geq 0 \;\; \forall S_0..S_n : (S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_n) \Rightarrow (S_0.M_c = S_n.M_c)$

*Proof* Examining both normal and attack step cases of $S \rightarrow S'$, we always find that $S.M_c = S'.M_c$. So, $S_0.M_c = S_1.M_c$, $S_1.M_c = S_2.M_c$, ..., $S_{n-1}.M_c = S_n.M_c$. By the transitivity of equality, $S_0.M_c = S_n.M_c$. ■

**Corollary 4**
$\forall n \geq 0 \;\; \forall S_0..S_n : (S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_n \wedge I(S_0.M_c)) \Rightarrow I(S_n.M_c)$

**Lemma 5**
$\forall n \geq 0 \;\; \forall S_0..S_{n+1} :$
$$\left( \begin{array}{l} I(S_0.M_c) \;\; \wedge \\ S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{n+1} \;\; \wedge \\ \forall k \in \{0..(n-1)\} : \mathrm{TB}(S_k, S_{k+1}) \Rightarrow \neg \mathrm{MIC_s}(S_{k+1}.M_c, S_{k+1}.pc) \\ \Rightarrow \\ \mathrm{TB}(S_n, S_{n+1}) \Rightarrow \neg \mathrm{MIC_s}(S_{n+1}.M_c, S_{n+1}.pc) \;\; \wedge \\ (S_n \rightarrow_n S_{n+1}) \Rightarrow S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc) \end{array} \right)$$

*Proof* There are two high-level cases to consider; either $S_n \not\rightarrow_n S_{n+1}$ or $S_n \rightarrow_n S_{n+1}$. If $S_n \not\rightarrow_n S_{n+1}$ then the definition of the predicate TB implies that $\neg \mathrm{TB}(S_n, S_{n+1})$.

We break up the case where $S_n \rightarrow_n S_{n+1}$ into several subcases depending on the type of instruction pointed to by the program counter in state $S_n$.

- Case $Dc(S_n.M_c(S_n.pc)) = jmp \;\; r_s$:
  By the definition of the TB predicate, $\mathrm{TB}(S_n, S_{n+1}) \Rightarrow S_n \rightarrow_n S_{n+1}$, so we need only assume that $S_n \rightarrow_n S_{n+1}$ and show that $\neg \mathrm{MIC_s}(S_{n+1}.M_c, S_{n+1}.pc)$ and $S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc)$.

  Constraint [I-Jmp] on $S_n.M_c$ (applicable because Corollary 4 implies that $I(S_n.M_c)$) requires $S_n.pc - 5$ to be a valid memory address in $S_n.M_c$. By definition, code memory begins at address 0, so $S_n.pc \geq 5$. Combining this with the facts that $S_0.pc = 0$ and $S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_n$ and attack steps cannot change the $pc$, we find that there must be at least one normal step between $S_0$ and $S_n$. So, $\exists i \in \{0..(n-1)\} : S_0 \rightarrow ... \rightarrow S_i \rightarrow_n S_{i+1} \rightarrow_a ... \rightarrow_a S_n$, where $S_{i+1}.pc = S_n.pc$ and $S_{i+1}.M_c = S_n.M_c$ because attack steps cannot alter the program counter or code memory. We have by assumption that $\mathrm{TB}(S_i, S_{i+1}) \Rightarrow \neg \mathrm{MIC_s}(S_{i+1}.M_c, S_{i+1}.pc)$, but $\mathrm{MIC_s}(S_{i+1}.M_c, S_{i+1}.pc)$, so contrapositively, $\neg \mathrm{TB}(S_i, S_{i+1})$. Examining cases of normal step semantics where $S_i \rightarrow_n S_{i+1}$ but $\neg \mathrm{TB}(S_i, S_{i+1})$, we find that $S_i.pc = S_{i+1}.pc - 1$, or equivalently, $S_i.pc = S_n.pc - 1$.

  Because $S_n.pc \geq 5$ and $S_i.pc = S_n.pc - 1$, we can apply arguments analogous to those of the previous paragraph to show that $\exists S_h \exists S_i : S_0 \rightarrow ... \rightarrow S_h \rightarrow_n S_{h+1} \rightarrow_a ... \rightarrow_a S_i \rightarrow_n S_{i+1} \rightarrow_a ... \rightarrow_a S_n$ and $S_h.pc = S_i.pc - 1$. Continuing similarly, we eventually find that $\exists S_e..S_i : S_0 \rightarrow ... \rightarrow S_e \rightarrow_n S_{e+1} \rightarrow_a ... \rightarrow_a S_f \rightarrow_n S_{f+1} \rightarrow_a$

$\ldots \rightarrow_a S_g \rightarrow_n S_{g+1} \rightarrow_a \ldots \rightarrow_a S_h \rightarrow_n S_{h+1} \rightarrow_a \ldots \rightarrow_a S_i \rightarrow_n S_{i+1} \rightarrow_a \ldots \rightarrow_a S_n$ and $S_e.pc = S_n.pc - 5 \wedge S_f.pc = S_n.pc - 4 \wedge \ldots \wedge S_i.pc = S_n.pc - 1$. Also, because the attack steps cannot alter the code memory, $pc$, or distinguished registers (by rule (As-Step)), $S_{e+1}.M_c = S_f.M_c \wedge S_{e+1}.pc = S_f.pc \wedge S_{e+1}.R_{0-2} = S_f.R_{0-2}$, etc.

Now, we let $S_e.R_{0-2} = R\{r_0 \mapsto a_0, r_1 \mapsto a_1, r_2 \mapsto a_2\}$ and trace execution according to the steps in the previous paragraph.

- Because $I(S_n.M_c)$, [I-Jmp] implies that $\exists r'_s : Dc(S_n.M_c(S_n.pc{-}5)) = addi\ r_0, r'_s, 0$. Combining this with the facts that $S_e.M_c = S_n.M_c$ (by Lemma 3), $S_e.pc = S_n.pc - 5$, and $S_e \rightarrow_n S_{e+1}$, we have $S_{e+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto a_1, r_2 \mapsto a_2\}$, where $a_s = S_e.R(r'_s)$.

Continuing similarly,

- $S_{f+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto a_2\}$.
- $\exists w_1 : S_{g+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto w_1\} \wedge \exists w_2 : Dc(w_1) = nop\ w_2 \wedge \forall a' \in \mathrm{dom}(S_n.M_c) : Dc(S_n.M_c(a')) = nop\ w_2 \Rightarrow a' \in \mathrm{succ}(S_n.M_c, S_n.pc)$.
- $S_{h+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge S_f.M(a_s) \leq w_1$ ($S_h$ cannot branch to *illegal* because $S_{h+1}.pc = S_i.pc$ and $S_i \rightarrow_n S_{i+1}$).
- $S_{i+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge S_f.M(a_s) = w_1$ ($S_i$ cannot branch to *illegal* because $S_{i+1}.pc = S_n.pc$ and $S_n \rightarrow_n S_{n+1}$).
- Finally, $S_n.R_{0-2} = S_{g+1}.R_{0-2} \wedge Dc(S_n.M_c(S_n.pc)) = jmp\ r_0$.

Combining the fact that $S_f.M(a_s) = w_1$ (by the second-to-last bullet above) with $Dc(w_1) = nop\ w_2$ (by the fourth-to-last bullet above), we obtain $Dc(S_f.M(a_s)) = nop\ w_2$. Also, because $S_n.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto w_1\}$ and $S_n \rightarrow_n S_{n+1}$ and $Dc(S_n.M_c(S_n.pc)) = jmp\ r_0$, we have by the normal-step rule for *jmp* instructions that $S_{n+1}.pc = a_s$ and $a_s \in \mathrm{dom}(S_n.M_c)$. Since $Dc(S_f.M(a_s)) = nop\ w_2$ (from above) and $S_f.M_c = S_n.M_c$ (by Lemma 3), we now have $Dc(S_n.M_c(a_s)) = nop\ w_2$. Therefore, the last clause of the fourth-to-last bullet above implies that $a_s = S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc)$. Finally, because $Dc(S_{n+1}.M_c(S_{n+1}.pc)) = nop\ w_2$, and $I(S_{n+1}.M_c)$, we have by the definition of the MIC predicate and the [I-Jmp] constraint on $S_{n+1}.M_c$ that $\neg\mathrm{MIC}(S_{n+1}.M_c, S_{n+1}.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = jd\ w$:
  By Corollary 4, $I(S_n.M_c)$. By the normal semantics of *jd* instructions and the definition of the TB predicate, $\mathrm{TB}(S_n, S_{n+1}) \Rightarrow S_{n+1}.pc = w$, so the [I-Jd] constraint on $S_n.M_c$ implies that $\mathrm{TB}(S_n, S_{n+1}) \Rightarrow \neg\mathrm{MIC}(S_{n+1}.M_c, S_{n+1}.pc)$ and $S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = bgt\ r_s, r_t, w$:
  By Corollary 4, $I(S_n.M_c)$. By the normal semantics of *bgt* instructions and the definition of the TB predicate, $\mathrm{TB}(S_n, S_{n+1}) \Rightarrow S_{n+1}.pc = w$, so the [I-Bgt] constraint on $S_n.M_c$ implies that $\mathrm{TB}(S_n, S_{n+1}) \Rightarrow \neg\mathrm{MIC}(S_{n+1}.M_c, S_{n+1}.pc)$. Finally, the [I-Bgt] and [I-Fall] constraints on $S_n.M_c$ ensure that $S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = illegal$:
  This case cannot occur because the language provides no normal transition from a state whose program counter points to an *illegal* instruction.

- Otherwise:
  $\neg\text{TB}(S_n, S_{n+1})$ because $S_n.M_c(S_n.pc)$ does not decode into a *jmp*, *jd*, or *bgt* instruction. Examination of the normal semantics for instructions that are not taken branches shows that $S_{n+1}.pc = S_n.pc + 1$, so by constraint [I-Fall] (applicable on $S_n.M_c$ by Corollary 4), $S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$.

■

**Lemma 6**

$$\forall n \geq 0 \;\; \forall S_0..S_{n+1} \;\; \forall i \in \{0..n\} : \left( \begin{array}{l} I(S_0.M_c) \;\; \wedge \\ S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{n+1} \\ \Rightarrow \\ \text{TB}(S_i, S_{i+1}) \Rightarrow \neg\text{MIC}(S_{i+1}.M_c, S_{i+1}.pc) \;\; \wedge \\ (S_i \rightarrow_n S_{i+1}) \Rightarrow S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc) \end{array} \right)$$

*Proof* By strong induction on i.

- Case $i = 0$:
  Because $S_i = S_0$, we have $I(S_i.M_c)$ and $S_i.pc = 0$. By the [I-Jmp] constraint on $S_i.M_c$ and the assumption that code begins at address 0 in $S_0.M_c$, $\forall r_s : Dc(S_i.M_c(S_i.pc)) \neq jmp\ r_s$. If $\text{TB}(S_i, S_{i+1})$, we use this fact to determine that $S_i$'s program counter points to either a *jd* or a *bgt* instruction. Because $I(S_i.M_c)$, we can then apply constraints [I-Jd] and [I-Bgt] to find that $\neg\text{MIC}(S_{i+1}.M_c, S_{i+1}.pc)$.

  Finally, we must show that if $S_i \rightarrow_n S_{i+1}$ then $S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)$, given that $\forall r_s : Dc(S_i.M_c(S_i.pc)) \neq jmp\ r_s$. Also note that $Dc(S_i.M_c(S_i.pc)) \neq illegal$ because the semantics prevent a normal step from a state in which the program counter addresses an *illegal* instruction. Examining the other types of instructions to which $S_i.pc$ can point, we find that the semantics for normal steps combined with the [I-Jd], [I-Bgt], and [I-Fall] constraints on $S_i.M_c$ always ensures that $S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)$.

- Case $i > 0$:
  By assumption, $I(S_0.M_c) \wedge S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{i+1}$. By the inductive hypothesis, $\forall k \in \{0..(i-1)\} : \text{TB}(S_k, S_{k+1}) \Rightarrow \neg\text{MIC}(S_{k+1}.M_c, S_{k+1}.pc)$. Combining these facts, we can apply Lemma 5 to obtain $\text{TB}(S_i, S_{i+1}) \Rightarrow \neg\text{MIC}(S_{i+1}.M_c, S_{i+1}.pc)$, and $(S_i \rightarrow_n S_{i+1}) \Rightarrow S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)$.

■

### A.3 Proof of Theorem 1

*Proof* There are two cases to consider; either $S_i \rightarrow_a S_{i+1}$ or $S_i \not\rightarrow_a S_{i+1}$. When $S_i \rightarrow_a S_{i+1}$, the semantics of attack steps implies that $S_{i+1}.pc = S_i.pc$. When $S_i \not\rightarrow_a S_{i+1}$, it must be

the case that $S_i \rightarrow_n S_{i+1}$, and the result is immediate from Lemma 6. ■

# B Further Formal Material on CFI with SMAC

## B.1 What Instrumentation Implies, Revisited

Much as in Section A.1, SMAC-based CFI instrumentation immediately leads to several static constraints on code memory. The definitions of the constraints and the analysis in the next subsection use the TB predicate (defined in Section A.1) and the $\text{MIC}_\text{s}$ predicate (defined below).

- $\text{MIC}_\text{s}(M_c, w) \iff \left( \begin{array}{l} \exists k \in \{0..8\} \exists r_s : Dc(M_c(w+k)) = jmp\ r_s \vee \\ \exists k \in \{0..4\} \exists r_d \exists w' \exists r_s : Dc(M_c(w+k)) = st\ r_d(w'), r_s \end{array} \right)$
  ($w$ is in the middle of instrumented code before a *jmp* or *st* instruction)

The static constraints on code memory induced by the instrumentation algorithm follow.

[$\text{I}_\text{s}$-Jmp] $\forall M_c\ \ \forall a \in \text{dom}(M_c)\ \ \forall r_s$ :

$$Dc(M_c(a)) = jmp\ r_s \Rightarrow \left( \begin{array}{l} \exists r'_s : Dc(M_c(a-9)) = addi\ r_0, r'_s, 0\ \ \wedge \\ \exists w_1 : Dc(M_c(a-8)) = movi\ r_1, w_1\ \ \wedge \\ \qquad w_1 = \max(M_c)\ \ \wedge \\ \exists w_2 : Dc(M_c(a-7)) = movi\ r_2, w_2\ \ \wedge \\ \qquad w_2 = \min(M_c)\ \ \wedge \\ \exists w_3 : Dc(M_c(a-6)) = bgt\ r_0, r_1, w_3\ \ \wedge \\ \qquad Dc(M_c(a-5)) = bgt\ r_1, r_0, w_3\ \ \wedge \\ \qquad Dc(M_c(w_3)) = illegal\ \ \wedge \\ Dc(M_c(a-4)) = ld\ r_1, r_0(0)\ \ \wedge \\ \exists w_4\ \exists w_5\ \forall a' \in \text{dom}(M_c) : \\ \qquad Dc(M_c(a-3)) = movi\ r_2, w_4\ \ \wedge \\ \qquad Dc(w_4) = nop\ w_5\ \ \wedge \\ \qquad Dc(M_c(a')) = nop\ w_5 \Rightarrow a' \in \text{succ}(M_c, a)\ \ \wedge \\ \exists w_6 : Dc(M_c(a-2)) = bgt\ r_1, r_2, w_6\ \ \wedge \\ \qquad Dc(M_c(a-1)) = bgt\ r_2, r_1, w_6\ \ \wedge \\ \qquad Dc(M_c(w_6)) = illegal\ \ \wedge \\ r_s = r_0 \end{array} \right)$$

[$\text{I}_\text{s}$-Jd] $\forall M_c\ \ \forall a \in \text{dom}(M_c)\ \ \forall w$ :

$$Dc(M_c(a)) = jd\ w \Rightarrow \left( \begin{array}{l} w \in \text{succ}(M_c, a)\ \wedge \\ \neg \text{MIC}_\text{s}(M_c, w) \end{array} \right)$$

[$\text{I}_\text{s}$-Bgt] $\forall M_c\ \ \forall a \in \text{dom}(M_c)\ \ \forall r_s\ \ \forall r_t\ \ \forall w$ :

$$Dc(M_c(a)) = bgt\ r_s, r_t, w \Rightarrow \left( \begin{array}{l} w \in \text{succ}(M_c, a)\ \ \wedge \\ \neg \text{MIC}_\text{s}(M_c, w) \end{array} \right)$$

[$\mathrm{I_s}$-Fall] $\forall M_c \ \forall a \in \mathrm{dom}(M_c)$ :

$$\left(\begin{array}{l} Dc(M_c(a)) \neq \mathit{illegal} \ \wedge \\ \forall r_s : Dc(M_c(a)) \neq \mathit{jmp}\ r_s \ \wedge \\ \forall w : Dc(M_c(a)) \neq \mathit{jd}\ w \end{array}\right) \Rightarrow a+1 \in \mathrm{succ}(M_c, a)$$

[$\mathrm{I_s}$-St] $\forall M_c \ \forall a \in \mathrm{dom}(M_c) \ \forall r_d \ \forall w \ \forall r_s$ :

$$Dc(M_c(a)) = \mathit{st}\ r_d(w), r_s \Rightarrow \left(\begin{array}{l} \exists r'_s : Dc(M_c(a-5)) = \mathit{addi}\ r_0, r'_s, 0 \ \wedge \\ \exists w_1 : Dc(M_c(a-4)) = \mathit{movi}\ r_1, w_1 \ \wedge \\ \qquad w_1 = \max(M_d) \ \wedge \\ \exists w_2 : Dc(M_c(a-3)) = \mathit{movi}\ r_2, w_2 \ \wedge \\ \qquad w_2 = \min(M_d) \ \wedge \\ \exists w_3 : Dc(M_c(a-2)) = \mathit{bgt}\ r_0, r_1, w_3 \ \wedge \\ \qquad Dc(M_c(a-1)) = \mathit{bgt}\ r_1, r_0, w_3 \ \wedge \\ \qquad Dc(M_c(w_3)) = \mathit{illegal} \ \wedge \\ r_d = r_0 \ \wedge \\ w = 0 \end{array}\right)$$

[$\mathrm{I_s}$-Ill] $\forall M_c : Dc(\max(M_c)) = \mathit{illegal}$

## B.2 Lemmas on SMAC-based CFI Enforcement

**Lemma 7**

$$\forall n \geq 0 \ \forall S_0..S_{n+1} : \left(\begin{array}{l} \mathrm{I_s}(S_0.M_c) \ \wedge \\ S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{n+1} \ \wedge \\ \forall j \in \{1..n\} : S_j.M_c = S_0.M_c \ \wedge \\ \forall k \in \{0..(n-1)\} : \mathrm{TB}(S_k, S_{k+1}) \Rightarrow \neg\mathrm{MIC_s}(S_{k+1}.M_c, S_{k+1}.pc) \\ \Rightarrow \\ S_{n+1}.M_c = S_0.M_c \ \wedge \\ \mathrm{TB}(S_n, S_{n+1}) \Rightarrow \neg\mathrm{MIC_s}(S_{n+1}.M_c, S_{n+1}.pc) \ \wedge \\ (S_n \rightarrow_n S_{n+1}) \Rightarrow S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc) \end{array}\right)$$

*Proof* There are two high-level cases to consider; either $S_n \not\rightarrow_n S_{n+1}$ or $S_n \rightarrow_n S_{n+1}$. If $S_n \not\rightarrow_n S_{n+1}$ then $S_n \rightarrow_a S_{n+1}$, and the semantics of attack steps immediately implies that $S_{n+1}.M_c = S_n.M_c$. By the assumption that $S_n.M_c = S_0.M_c$, we have $S_{n+1}.M_c = S_0.M_c$. In addition, the definition of the predicate TB implies that $\neg\mathrm{TB}(S_n, S_{n+1})$ when $S_n \not\rightarrow_n S_{n+1}$.

We break up the case where $S_n \rightarrow_n S_{n+1}$ into several subcases depending on the type of instruction pointed to by the program counter in state $S_n$.

- Case $Dc(S_n.M_c(S_n.pc)) = \mathit{st}\ r_d(w), r_s$:
  $\neg\mathrm{TB}(S_n, S_{n+1})$ because $S_n.M_c(S_n.pc)$ does not decode into a *jmp*, *jd*, or *bgt* instruction. In addition, the normal semantics for *st* instructions shows that $S_{n+1}.pc = S_n.pc+1$, so by constraint [$\mathrm{I_s}$-Fall] (applicable because $I(S_0.M_c)$ and $S_n.M_c = S_0.M_c$), $S_{n+1}.pc \in \mathrm{succ}(S_0.M_c, S_n.pc)$ when $S_n \rightarrow_n S_{n+1}$. When $S_n \rightarrow_a S_{n+1}$, the semantics of attack steps ensures that $S_{n+1}.M_c = S_0.M_c$, so all that remains is to show that $S_{n+1}.M_c = S_0.M_c$ under the assumption that $S_n \rightarrow_n S_{n+1}$.

  Constraint [$\mathrm{I_s}$-St] on $S_n.M_c$ requires $S_n.pc-5$ to be a valid memory address in $S_n.M_c$. By definition, code memory begins at address 0, so $S_n.pc \geq 5$. Combining this with

the facts that $S_0.pc = 0$ and $S_0 \to S_1 \to ... \to S_n$ and attack steps cannot change the $pc$, we find that there must be at least one normal step between $S_0$ and $S_n$. So, $\exists i \in \{0..(n-1)\} : S_0 \to ... \to S_i \to_n S_{i+1} \to_a ... \to_a S_n$, where $S_{i+1}.pc = S_n.pc$ and $S_{i+1}.M_c = S_n.M_c$ because attack steps cannot alter the program counter or code memory. We have by assumption that $\text{TB}(S_i, S_{i+1}) \Rightarrow \neg\text{MIC}_\text{s}(S_{i+1}.M_c, S_{i+1}.pc)$, but $\text{MIC}_\text{s}(S_{i+1}.M_c, S_{i+1}.pc)$, so contrapositively, $\neg\text{TB}(S_i, S_{i+1})$. Examining cases of normal step semantics where $S_i \to_n S_{i+1}$ but $\neg\text{TB}(S_i, S_{i+1})$, we find that $S_i.pc = S_{i+1}.pc - 1$, or equivalently, $S_i.pc = S_n.pc - 1$.

Because $S_n.pc \geq 5$ and $S_i.pc = S_n.pc - 1$, we can apply arguments analogous to those of the previous paragraph to show that $\exists S_h \exists S_i : S_0 \to ... \to S_h \to_n S_{h+1} \to_a ... \to_a S_i \to_n S_{i+1} \to_a ... \to_a S_n$ and $S_h.pc = S_i.pc - 1$. Continuing similarly, we eventually find that $\exists S_e..S_i : S_0 \to ... \to S_e \to_n S_{e+1} \to_a ... \to_a S_f \to_n S_{f+1} \to_a ... \to_a S_g \to_n S_{g+1} \to_a ... \to_a S_h \to_n S_{h+1} \to_a ... \to_a S_i \to_n S_{i+1} \to_a ... \to_a S_n$ and $S_e.pc = S_n.pc - 5 \wedge S_f.pc = S_n.pc - 4 \wedge ... \wedge S_i.pc = S_n.pc - 1$. Also, because the attack steps cannot alter the code memory, $pc$, or distinguished registers (by rule (As-Step)), $S_{e+1}.M_c = S_f.M_c \wedge S_{e+1}.pc = S_f.pc \wedge S_{e+1}.R_{0-2} = S_f.R_{0-2}$, etc.

Now, we let $S_e.R_{0-2} = R\{r_0 \mapsto a_0, r_1 \mapsto a_1, r_2 \mapsto a_2\}$ and trace execution according to the steps in the previous paragraph.

- Because $\text{I}_\text{s}(S_n.M_c)$, [$\text{I}_\text{s}$-St] implies that $\exists r'_s.Dc(S_n.M_c(S_n.pc-5)) = \mathit{addi}\ r_0, r'_s, 0$. Combining this with the facts that $S_e.M_c = S_n.M_c$, $S_e.pc = S_n.pc - 5$, and $S_e \to_n S_{e+1}$, we have $S_{e+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto a_1, r_2 \mapsto a_2\}$, where $a_s = S_e.R(r'_s)$.

Continuing similarly,

- $S_{f+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto \max(S_0.M_d), r_2 \mapsto a_2\}$.
- $S_{g+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto \max(S_0.M_d), r_2 \mapsto \min(S_0.M_d)\}$.
- $S_{h+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge a_s \leq \max(S_0.M_d)$ ($S_h$ cannot branch to *illegal* because $S_{h+1}.pc = S_i.pc$ and $S_i \to_n S_{i+1}$).
- $S_{i+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge \min(S_0.M_d) \leq a_s \leq \max(S_0.M_d)$ ($S_i$ cannot branch to *illegal* because $S_{i+1}.pc = S_n.pc$ and $S_n \to_n S_{n+1}$).
- Finally, $S_n.R_{0-2} = S_{g+1}.R_{0-2} \wedge Dc(S_n.M_c(S_n.pc)) = \mathit{st}\ r_0(0), r_s$.

Because $S_n.R(r_0) = a_s$, $\min(S_0.M_d) \leq a_s \leq \max(S_0.M_d)$, $Dc(S_n.M_c(S_n.pc)) = \mathit{st}\ r_0(0), r_s$, and $S_n \to_n S_{n+1}$, the normal semantics for *st* instructions ensures that $S_{n+1}.M_c = S_0.M_c$.

- Case $Dc(S_n.M_c(S_n.pc)) = \mathit{jmp}\ r_s$:
Only instructions of the form $\mathit{st}\ r_d(w), r_s$ can modify code memory, so $S_{n+1}.M_c = S_0.M_c$. Because $I(S_0.M_c)$ and $S_n.M_c = S_0.M_c$, $I(S_n.M_c)$. By the definition of the TB predicate, $\text{TB}(S_n, S_{n+1}) \Rightarrow S_n \to_n S_{n+1}$, so we need only assume that $S_n \to_n S_{n+1}$ and show that $\neg\text{MIC}_\text{s}(S_{n+1}.M_c, S_{n+1}.pc)$ and $S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$.

  Constraint [$\text{I}_\text{s}$-Jmp] on $S_n.M_c$ requires $S_n.pc - 9$ to be a valid memory address in $S_n.M_c$. By definition, code memory begins at address 0, so $S_n.pc \geq 9$. Combining

this with the facts that $S_0.pc = 0$ and $S_0 \to S_1 \to ... \to S_n$ and attack steps cannot change the $pc$, we find that there must be at least one normal step between $S_0$ and $S_n$. So, $\exists i \in \{0..(n-1)\} : S_0 \to ... \to S_i \to_n S_{i+1} \to_a ... \to_a S_n$, where $S_{i+1}.pc = S_n.pc$ and $S_{i+1}.M_c = S_n.M_c$ because attack steps cannot alter the program counter or code memory. We have by assumption that $\text{TB}(S_i, S_{i+1}) \Rightarrow \neg\text{MIC}_\text{s}(S_{i+1}.M_c, S_{i+1}.pc)$, but $\text{MIC}_\text{s}(S_{i+1}.M_c, S_{i+1}.pc)$, so contrapositively, $\neg\text{TB}(S_i, S_{i+1})$. Examining cases of normal step semantics where $S_i \to_n S_{i+1}$ but $\neg\text{TB}(S_i, S_{i+1})$, we find that $S_i.pc = S_{i+1}.pc - 1$, or equivalently, $S_i.pc = S_n.pc - 1$.

Because $S_n.pc \geq 9$ and $S_i.pc = S_n.pc - 1$, we can apply arguments analogous to those of the previous paragraph to show that $\exists S_h \exists S_i : S_0 \to ... \to S_h \to_n S_{h+1} \to_a ... \to_a S_i \to_n S_{i+1} \to_a ... \to_a S_n$ and $S_h.pc = S_i.pc - 1$. Continuing similarly, we eventually find that $\exists S_a..S_i : S_0 \to ... \to S_a \to_n S_{a+1} \to_a ... \to_a S_b \to_n S_{b+1} \to_a ... \to_a S_c \to_n S_{c+1} \to_a ... \to_a S_d \to_n S_{d+1} \to_a ... \to_a S_e \to_n S_{e+1} \to_a ... \to_a S_f \to_n S_{f+1} \to_a ... \to_a S_g \to_n S_{g+1} \to_a ... \to_a S_h \to_n S_{h+1} \to_a ... \to_a S_i \to_n S_{i+1} \to_a ... \to_a S_n$ and $S_a.pc = S_n.pc - 9 \wedge S_b.pc = S_n.pc - 8 \wedge ... \wedge S_i.pc = S_n.pc - 1$. Also, because the attack steps cannot alter the code memory, $pc$, or distinguished registers (by rule (As-Step)), $S_{a+1}.M_c = S_b.M_c \wedge S_{a+1}.pc = S_b.pc \wedge S_{a+1}.R_{0-2} = S_b.R_{0-2}$, etc.

Now, we let $S_a.R_{0-2} = R\{r_0 \mapsto a_0, r_1 \mapsto a_1, r_2 \mapsto a_2\}$ and trace execution according to the steps in the previous paragraph, analogously to the execution trace in the previous case (for *st* instructions).

- $S_{a+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto a_1, r_2 \mapsto a_2\}$, where $a_s = S_a.R(r'_s)$.
- $S_{b+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto \max(S_0.M_c), r_2 \mapsto a_2\}$.
- $S_{c+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto \max(S_0.M_c), r_2 \mapsto \min(S_0.M_c)\}$.
- $S_{d+1}.R_{0-2} = S_{c+1}.R_{0-2} \wedge a_s \leq \max(S_0.M_c)$ ($S_d$ cannot branch to *illegal* because $S_{d+1}.pc = S_e.pc$ and $S_e \to_n S_{e+1}$).
- $S_{e+1}.R_{0-2} = S_{c+1}.R_{0-2} \wedge \min(S_0.M_c) \leq a_s \leq \max(S_0.M_c)$ ($S_e$ cannot branch to *illegal* because $S_{e+1}.pc = S_f.pc$ and $S_f \to_n S_{f+1}$).
- $S_{f+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto \min(S_0.M_c)\}$.
- $\exists w_1 : S_{g+1}.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto w_1\} \wedge \exists w_2 : Dc(w_1) = nop\ w_2 \wedge \forall a' \in \text{dom}(S_n.M_c) : Dc(S_n.M_c(a')) = nop\ w_2 \Rightarrow a' \in \text{succ}(S_n.M_c, S_n.pc)$.
- $S_{h+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge S_f.M(a_s) \leq w_1$ ($S_h$ cannot branch to *illegal* because $S_{h+1}.pc = S_i.pc$ and $S_i \to_n S_{i+1}$).
- $S_{i+1}.R_{0-2} = S_{g+1}.R_{0-2} \wedge S_f.M(a_s) = w_1$ ($S_i$ cannot branch to *illegal* because $S_{i+1}.pc = S_n.pc$ and $S_n \to_n S_{n+1}$).
- Finally, $S_n.R_{0-2} = S_{g+1}.R_{0-2} \wedge Dc(S_n.M_c(S_n.pc)) = jmp\ r_0$.

Combining the fact that $S_f.M(a_s) = w_1$ (by the second-to-last bullet above) with $Dc(w_1) = nop\ w_2$ (by the fourth-to-last bullet above), we obtain $Dc(S_f.M(a_s)) = nop\ w_2$. Also, because $S_n.R_{0-2} = R\{r_0 \mapsto a_s, r_1 \mapsto S_f.M(a_s), r_2 \mapsto w_1\}$ and $S_n \to_n S_{n+1}$ and $Dc(S_n.M_c(S_n.pc)) = jmp\ r_0$, we have by the normal-step rule for *jmp* instructions that $S_{n+1}.pc = a_s$. Moreover, $a_s \in \text{dom}(S_n.M_c)$ by the fifth bullet above and the fact that $S_0.M_c = S_n.M_c$. Since $Dc(S_f.M(a_s)) = nop\ w_2$ (from

above), we now have $Dc(S_n.M_c(a_s)) = nop\ w_2$. Therefore, the last clause of the fourth-to-last bullet above implies that $a_s = S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$. Finally, because $Dc(S_{n+1}.M_c(S_{n+1}.pc)) = nop\ w_2$, and $\text{I}_\text{s}(S_{n+1}.M_c)$, we have by the definition of the $\text{MIC}_\text{s}$ predicate and the [$\text{I}_\text{s}$-Jmp] and [$\text{I}_\text{s}$-St] constraints on $S_{n+1}.M_c$ that $\neg\text{MIC}_\text{s}(S_{n+1}.M_c, S_{n+1}.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = jd\ w$:
  Only instructions of the form $st\ r_d(w), r_s$ can modify code memory, so $S_{n+1}.M_c = S_0.M_c$. Because $I(S_0.M_c)$ and $S_n.M_c = S_0.M_c$, $I(S_n.M_c)$. By the normal semantics of *jd* instructions and the definition of the TB predicate, $\text{TB}(S_n, S_{n+1}) \Rightarrow S_{n+1}.pc = w$, so the [$\text{I}_\text{s}$-Jd] constraint on $S_n.M_c$ implies that $\text{TB}(S_n, S_{n+1}) \Rightarrow \neg\text{MIC}_\text{s}(S_{n+1}.M_c, S_{n+1}.pc)$ and $S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = bgt\ r_s, r_t, w$:
  Only instructions of the form $st\ r_d(w), r_s$ can modify code memory, so $S_{n+1}.M_c = S_0.M_c$. Because $I(S_0.M_c)$ and $S_n.M_c = S_0.M_c$, $I(S_n.M_c)$. By the normal semantics of *bgt* instructions and the definition of the TB predicate, $\text{TB}(S_n, S_{n+1}) \Rightarrow S_{n+1}.pc = w$, so the [$\text{I}_\text{s}$-Bgt] constraint on $S_n.M_c$ implies that $\text{TB}(S_n, S_{n+1}) \Rightarrow \neg\text{MIC}_\text{s}(S_{n+1}.M_c, S_{n+1}.pc)$. Finally, the [$\text{I}_\text{s}$-Bgt] and [$\text{I}_\text{s}$-Fall] constraints on $S_n.M_c$ ensure that $S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$.

- Case $Dc(S_n.M_c(S_n.pc)) = illegal$:
  This case cannot occur because the language provides no normal transition from a state whose program counter points to an *illegal* instruction.

- Otherwise:
  Only instructions of the form $st\ r_d(w), r_s$ can modify code memory, so $S_{n+1}.M_c = S_0.M_c$. In addition, $\neg\text{TB}(S_n, S_{n+1})$ because $S_n.M_c(S_n.pc)$ does not decode into a *jmp*, *jd*, or *bgt* instruction. Finally, examination of the normal semantics for instructions that are not taken branches shows that $S_{n+1}.pc = S_n.pc + 1$, so by constraint [$\text{I}_\text{s}$-Fall] (applicable because $\text{I}_\text{s}(S_0.M_c)$ and $S_n.M_c = S_0.M_c$), $S_{n+1}.pc \in \text{succ}(S_0.M_c, S_n.pc)$.

■

**Lemma 8**

$$\forall n \geq 0\ \ \forall S_0..S_{n+1}\ \ \forall i \in \{0..n\} : \left( \begin{array}{l} \text{I}_\text{s}(S_0.M_c)\ \ \wedge \\ S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{n+1} \\ \Rightarrow \\ S_{i+1}.M_c = S_0.M_c\ \ \wedge \\ \text{TB}(S_i, S_{i+1}) \Rightarrow \neg\text{MIC}_\text{s}(S_{i+1}.M_c, S_{i+1}.pc)\ \ \wedge \\ (S_i \rightarrow_n S_{i+1}) \Rightarrow S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc) \end{array} \right)$$

*Proof* By strong induction on i.

- Case $i = 0$:
  Because $S_i = S_0$, we have $\text{I}_\text{s}(S_i.M_c)$ and $S_i.pc = 0$. By the [$\text{I}_\text{s}$-Jmp] and [$\text{I}_\text{s}$-St] constraints on $S_i.M_c$ and the assumption that code begins at address 0 in $S_0.M_c$, $\forall r_s : Dc(S_i.M_c(S_i.pc)) \neq jmp\ r_s$ and $\forall r_d\ \forall w\ \forall r_s : Dc(S_i.M_c(S_i.pc)) \neq st\ r_d(w), r_s$.

By the semantics of attack and normal steps, only a normal step from a state whose program counter points to a *st* instruction can change code memory. Therefore, $\forall r_d\ \forall w\ \forall r_s : Dc(S_i.M_c(S_i.pc)) \neq st\ r_d(w), r_s$ implies that $S_{i+1}.M_c = S_0.M_c$.

If $\mathrm{TB}(S_i, S_{i+1})$, we use the fact that $\forall r_s : Dc(S_i.M_c(S_i.pc)) \neq jmp\ r_s$ to find that $S_i$'s program counter points to either a *jd* or a *bgt* instruction. Because $\mathrm{I_s}(S_i.M_c)$, we can then apply constraints [$\mathrm{I_s}$-Jd] and [$\mathrm{I_s}$-Bgt] to find that $\neg\mathrm{MIC_s}(S_{i+1}.M_c, S_{i+1}.pc)$.

Finally, we must show that if $S_i \rightarrow_n S_{i+1}$ then $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, S_i.pc)$, given that $\forall r_s : Dc(S_i.M_c(S_i.pc)) \neq jmp\ r_s$ and $\forall r_d\ \forall w\ \forall r_s : Dc(S_i.M_c(S_i.pc)) \neq st\ r_d(w), r_s$. Also note that $Dc(S_i.M_c(S_i.pc)) \neq illegal$ because the semantics prevent a normal step from a state in which the program counter addresses an *illegal* instruction. Examining the other types of instructions to which $S_i.pc$ can point, we find that the semantics for normal steps combined with the [$\mathrm{I_s}$-Jd], [$\mathrm{I_s}$-Bgt], and [$\mathrm{I_s}$-Fall] constraints on $S_i.M_c$ always ensures that $S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, S_i.pc)$.

- Case $i > 0$:
  By assumption, $\mathrm{I_s}(S_0.M_c) \wedge S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_{i+1}$. By the inductive hypothesis, $\forall j \in \{0..(i-1)\} : S_{j+1}.M_c = S_0.M_c$ and $\forall k \in \{0..(i-1)\} : \mathrm{TB}(S_k, S_{k+1}) \Rightarrow \neg\mathrm{MIC_s}(S_{k+1}.M_c, S_{k+1}.pc)$. Combining these facts, we can apply Lemma 7 to obtain $S_{i+1}.M_c = S_0.M_c$, $\mathrm{TB}(S_i, S_{i+1}) \Rightarrow \neg\mathrm{MIC_s}(S_{i+1}.M_c, S_{i+1}.pc)$, and $(S_i \rightarrow_n S_{i+1}) \Rightarrow S_{i+1}.pc \in \mathrm{succ}(S_0.M_c, S_i.pc)$.

■

### B.3 Proof of Theorem 2

*Proof* There are two cases to consider; either $S_i \rightarrow_a S_{i+1}$ or $S_i \not\rightarrow_a S_{i+1}$. When $S_i \rightarrow_a S_{i+1}$, the semantics of attack steps implies that $S_{i+1}.pc = S_i.pc$. When $S_i \not\rightarrow_a S_{i+1}$, it must be the case that $S_i \rightarrow_n S_{i+1}$, and the result is immediate from Lemma 8. ■